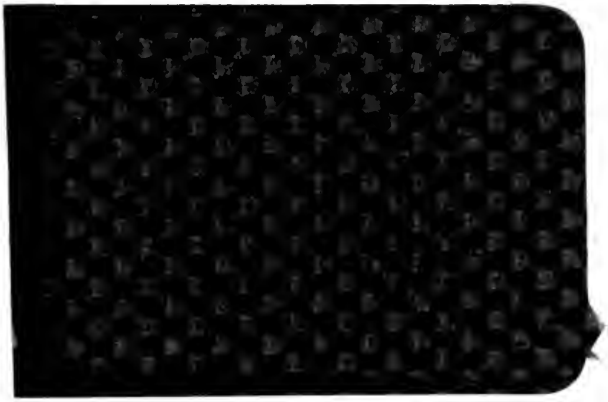
Computer Science

TECHNICAL REPORT

SMARTS — Shared-memory Multiprocessor Ada Run Time Supervisor

Technical Report 495



SMARTS – Shared-memory Multiprocessor Ada Run Time Supervisor

Susan Flynn Hummel

Technical Report 495

February 1990

SMARTS – Shared-memory Multiprocessor Ada Run Time Supervisor

Susan Flynn Hummel

Research Advisor: *Edmond Schonberg*

Abstract

The programming language Ada is primarily intended for the construction of large-scale and real-time systems. Although the tasking model of Ada was mainly aimed at embedded systems, its rich set of tasking features together with its support for programming in the large make Ada increasingly attractive for writing parallel programs for computationally-intensive numeric and symbolic applications.

Highly parallel shared-memory MIMD machines such as the NYU Ultracomputer have traditionally been regarded as suitable for large-scale scientific code, and not for more symbolic or heterogeneous concurrent applications such as are found in artificial intelligence or real-time programming. However, these applications would benefit greatly from (and even require) the computational power provided by machines with hundreds of processors.

It is therefore desirable to develop Ada implementations for highly parallel machines. The concern has been that the cost of managing large numbers of Ada tasks will negate the speedup obtained from their parallel execution. Indeed, a run-time supervisor for Ada must contend with many potentially expensive serialization points, that is, constructs that may take time proportional to the number of tasks involved.

In this thesis we show that a run-time supervisor for an implementation of Ada on highly parallel machines can be written which is free of costly serialization points. The run-time supervisor SMARTS (Shared-memory Multiprocessor Ada Run Time Supervisor) depends on the hardware synchronization primitive *fetch&*ϕ, and supports the tasking features of Ada in a highly parallel manner.

We further reduce the overhead of Ada tasking, by means of micro-tasking, i.e., the

explicit scheduling of a family of Ada tasks on a specified number of processors. Thus, Ada tasks are implemented as *lightweight* processes managed by SMARTS, rather than full-blown operating systems processes.

Finally, SMARTS implements Ada shared variables efficiently by means of *relay sets*. Relay sets provide a means not only for efficiently storing shared variables in a hierarchy of memories, but also for identifying shared variables. Furthermore, relays sets facilitate optimizations wherein rendezvous are executed in the environment of the calling task.

Acknowledgements

Many people contributed to this thesis. The most influential of these was my advisor, Ed Schonberg. Through numerous discussions and revisions of drafts, he not only greatly improved the content of this thesis, but vastly improved its style as well. Other people on the Ada project (most notably Robert Dewar) and the Ultracomputer project (most notably Jan Edler, Jim Lipkis, and Edith Schonberg) contributed technical material. The enjoyable summer I spent at IBM working for Fran Allen helped me to put my work in perspective. It was also a pleasure to work with Anne Dinning and Mike Hind on the implementation of SMARTS. Kurt Behnke carefully proofed a final draft.

My family, as always, were supportive throughout. Many friends provided encouragement, especially Bob who picked up the slack during the final push.

Table of Contents

1. Introduction

With the proliferation of multiprocessors it is becoming increasingly common for application programs to be executed in parallel. The parallel execution of a program is desirable since it may dramatically reduce its elapsed execution time. It is hoped that currently unpracticable computationally-intensive applications will be made practicable by their execution on highly parallel machines (multiprocessors with hundreds, if not thousands, of processors). Candidates for execution on these machines include large-scale scientific code and compute-bound artificial intelligence systems. Although not necessarily compute-bound, software for controlling large embedded systems (e.g., a telephone switching network) are also candidates for execution on highly parallel machines. These types of applications must perform parallel tasks and meet stringent real-time deadlines. To meet these deadlines, both the parallel execution and the computational power provided by highly parallel machines may be required.

To be executed on multiprocessors, application programs must be divided into subtasks that can be executed in parallel. The parallel subtasks of a program can either be detected by a compiler with rigorous program analysis (as in [Allen *et al.* 1987] and [Chen and Ci 1987]) or be identified by the programmer. Having a compiler detect the parallel subtasks of a program may be problematic for several reasons. Exhaustive analysis is expensive, and therefore, not practical when there are large numbers of processors and (potential) subtasks. Partial analysis may fail to detect all of the available parallelism. Further, even when all of the inherent parallelism in a sequential program is detected, the results may still be less than optimal since the optimal parallel algorithm for a given problem may bear little resemblance to the optimal (parallelized) sequential algorithm. Thus, to make effective use of highly parallel machines, given the state of the art in automatic detection of parallelism, we must rely on programmers to identify the parallel subtasks of a program, i.e., to write parallel programs.

To write a parallel program in a programming language without explicit parallel constructs, a programmer must extend the language with calls to operating system functions to create subtasks, synchronize subtasks etc. This is not only taxing on the programmer, but prone to error. Moreover, operating system calls may be expensive (in terms of time), and hence, negate the speedup obtained from parallelization. Similarly, programming languages without mechanisms for responding to software events or hardware interrupts within a given time of their occurrence force the writers of real-time application programs to use operating system functions to meet timing constraints, which again, is taxing on the programmer and error prone.

We will use the terms *parallel features* and *real-time features* to denote programming language constructs for controlling the parallel execution and expressing the timing constraints of a program (respectively). Programming languages that include high-level parallel and real-time features facilitate the writing of parallel and real-time programs for multiprocessors. By providing high-level constructs to express parallelism and timing, the burden of mapping parallel programs to the multiprocessor architecture and of managing the parallel subtasks is moved from the programmer to the compiler. The compiler, however, must be tailored to the particular multiprocessor architecture. To make the compiler more portable, the code for run-time support functions, such as a subtask scheduler, can be encapsulated in a package. Calls to this package, which is called a run-time supervisor (RTS), are then inserted into compiled programs. To retarget the compiler to a new machine only the code generator and the machine dependent portion of the RTS (the *kernel* of the RTS) need to be rewritten.

RTS's for sequential programming languages on uniprocessors are well understood. Indeed, the generation of RTS's for sequential languages has even been automated [Kaiser

1986]. In contrast, the RTS's of parallel languages are still very much a topic of research. The main function of an RTS for a sequential language is to manage data RTS's for parallel languages on uniprocessors are more complex. They must not only manage data, but the parallel subtasks as well. RTS's for parallel languages on multiprocessors are yet more complex. They must manage data stored in multiple memories and parallel subtasks executing on multiple processors. RTS's for real-time languages on multiprocessors are more complex still. They must ensure that a subtask begins executing (perhaps, on a specified processor) in response to a software event or hardware interrupt (that may have occurred on a different processor) within a guaranteed amount of time.

The speedup obtained by executing the optimal sequential program for an application on a multiprocessor is at most linear in the number of processors [Schwartz 1980]. To make effective use of increasing numbers of processors, an application program must be divided into increasing numbers of parallel subtasks. As the number of parallel subtasks increases, however, so does the work that must be performed by the RTS. The RTS overhead per subtask will determine the smallest granularity of subtask that it is worth executing in parallel (A method for determining the optimal subtask granularity is given in [Cytron 1985].). RTS functions that take time (at least) proportional to the number of parallel subtasks involved degrade performance by creating *serial bottlenecks*. When the RTS overhead per subtask increases more than linearly, a point will be reached where each additional subtask will incur a larger RTS overhead than it will speedup the computation, i e , dividing an application program into more parallel subtasks will actually slow down its overall elapsed execution time It is, therefore, imperative that RTS's for highly parallel machines be as far as possible free of serial bottlenecks For real time programming languages, where the functions performed by the RTS must meet real time deadlines, an efficient RTS is even more critical

Ada is a block structured parallel real time language Because of its rich set of parallel and real time features and its support for programming in the large, Ada is ideally suited for developing the programs necessary to make effective use of highly parallel machines. However, Ada has been perceived as being too richly endowed with features to admit an efficient implementation on this class of machines. Specifically, it has been assumed that the overhead of an Ada RTS will severely limit the potential speedup of Ada programs obtainable from their highly parallel execution.

In this thesis we show that an efficient RTS for Ada on highly parallel machines can be realized. To wit, we describe the implementation of such a RTS, called SMARTS: a Shared-memory Multiprocessor Ada RTS. Before giving the details of this implementation, we lay some technical groundwork. This material constitutes the rest of the chapter and is organized as follows: First, we present some general material regarding parallel machines and parallel real-time languages. We then give an overview of the Ada tasking model. Next, we consider in more detail the suitability of Ada for writing application programs for highly parallel machines. We conclude that it is only concerns about run-time performance that make the desirability of Ada implementations on highly parallel machines questionable. The functions that have to be performed by an RTS Ada on highly parallel machines are then identified. The rest of this thesis describes how SMARTS efficiently implements these functions. We finish the chapter with a brief history of SMARTS.

1.1 Nomenclature

To put both Ada and our implementation of it on highly parallel machines in perspective, we review some background material on parallel machines and languages We begin this section with a taxonomy of parallel architectures. The features of parallel and real-time programming languages are then discussed

1.1.1 Parallel Architectures

Multiprocessors consist of processors, memory modules and an interconnection structure. The interconnection structure connects processors to other processors or to memory modules. Multiprocessors can be categorized as *tightly-coupled* or *loosely-coupled*. When all of the processors are connected to all of the memory modules, i.e., the processors share a global memory, the multiprocessor is tightly-coupled. When the processors and memory modules are paired off, i.e., processors have local memories, and when all of the processors are connected to all of the other processors, the multiprocessor is loosely-coupled.

Many highly parallel tightly-coupled multiprocessors have been built or proposed, such as the Butterfly [BBN 1986], the Monarch [Rettburg 1987], and the Ultracomputer [Gottlieb 1987]. Loosely-coupled multiprocessors that have been built or proposed include the Connection Machine [Hillis 1985], the Cosmic Cube [Seitz 1985], and PASM [Siegel *et al.* 1981]. Figures 1.1.1 and 1.1.2 depict possible interconnection networks for tightly-coupled and loosely-coupled machines. The Omega network in figure 1.1.1 is used by the Ultracomputer. The hypercube in figure 1.1.2 is used by the Cosmic Cube. Note that both of these interconnection topologies allow processors to communicate with any memory module or processor in time that is logarithmic in the number of processors.

Hybrid architectures such as the RP3 [Pfister *et al.* 1985] are also becoming common. In a hybrid architecture each processor is paired with a memory module, but is also connected to all of the other memory modules. Thus, there are both local and global memories. The interconnection network of the RP3 is the same as that of the Ultracomputer (an Omega network), however, the interconnection network of the RP3 connects processor/memory pairs to other processor/memory pairs (rather than processors to memories). From a processor/memory/network configuration standpoint, the architectures of some loosely-coupled machines closely resemble the architectures of hybrid machines, for example, PASM

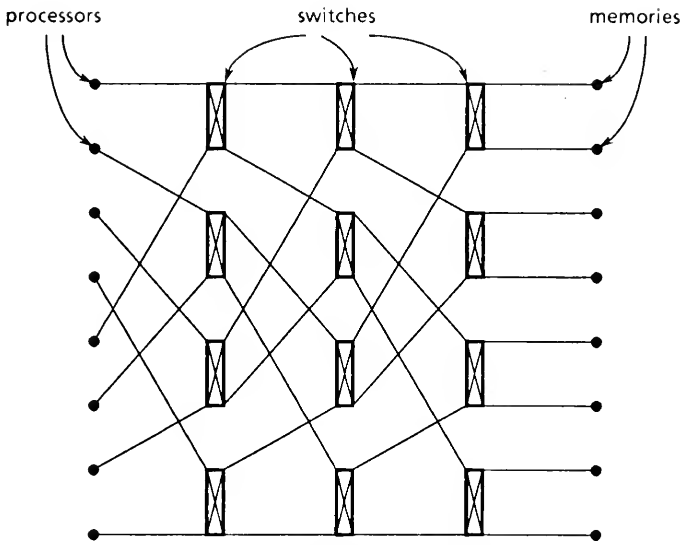


Figure 1.1.1: An Omega interconnection network

and the RP3. The processors of tightly-coupled multiprocessors may have local caches (e.g., the Ultracomputer), making them similar to hybrid architectures.

Because their processors share a global memory, tightly-coupled and hybrid machines are often called *shared-memory* machines. Because their processor/memory pairs are distributed, loosely-coupled machines are often called *distributed* machines. The processors of tightly-coupled machines can communicate via shared variables since the processors share a global address space. Whereas, the processors of loosely-coupled machines must communicate by passing messages.

A further distinction among multiprocessors is based on the instructions and data that the processors use. The processors may be controlled either by separate (multiple) instruction streams or by a single instruction stream. Similarly, the processors may operate either on multiple data or on a single datum. Multiple instruction multiple data (MIMD) and single instruction multiple data (SIMD) are the two most common types of multiprocessor

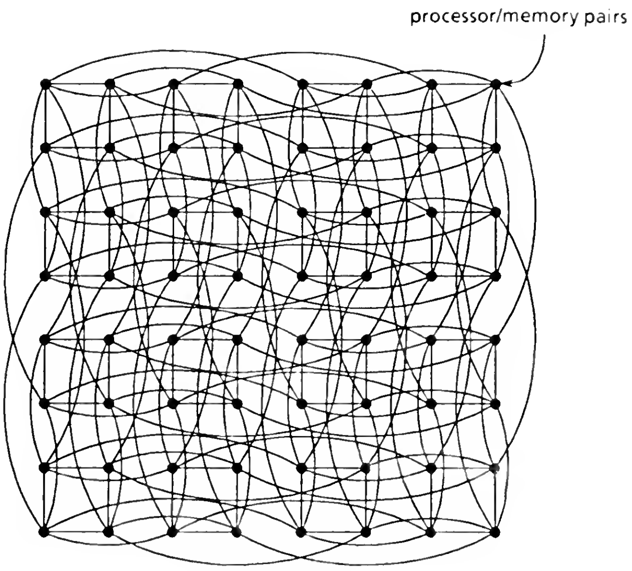


Figure 1.1.2: A Hypercube interconnection network

architectures. The Butterfly, Cosmic Cube, Monarch, Ultracomputer, RP3 are all MIMD machines. The Connection Machine is a SIMD machine. PASM can switch dynamically from SIMD to MIMD mode.

Most SIMD machines execute instructions in lock step and there is no need to synchronize the processors. The processors of some MIMD machines, such as the Monarch, also run synchronously. When the processors of a multiprocessor run asynchronously, hardware support for synchronizing the processors must be provided. Inter-processor synchronization can be achieved using the same mechanism as inter-processor communication. message passing for loosely-coupled machines and shared variables for tightly-coupled machines. *Inter-processor interrupts* are hardware mechanisms that enable the processors of loosely-coupled machines to synchronize. When a processor is sent a message, an interrupt is issued to notify the processor of the message's arrival. The synchronization primitives of tightly-coupled machines often involve some form of atomic *read-modify-write* instruction which operates on shared variables. For instance, the hardware synchronization primitive of the Ultracomputer is the fetch&ϕ instruction, where ϕ is a binary associative operator. Fetch&$\phi(v, e)$ indivisibly fetches the present value of v, performs $\phi(v, e)$ and stores this result in v.

A multiprocessor that contains diverse, perhaps special-purpose processors is said to be *heterogeneous*. When all of the processors that make up the multiprocessor are identical, the multiprocessor is said to be *homogeneous*. SMARTS is targeted to tightly-coupled, MIMD, homogeneous multiprocessors. However, with some slight (and not so slight) modifications SMARTS could be retargetted to other types of multiprocessors. (In Chapter 6 we consider these modifications in more detail.) For example, support for global name spaces, i.e., shared variables, has been implemented on various loosely-coupled machines (see, for example, [Basiani and Forin 1987a], [Basiani and Forin 1987b], [Carriero 1987], [Chiteron 1984], [Chiteron 1985], [Gelernter 1985], [LeBlanc 1983] and [Li 1986]). SMARTS could be ported to a distributed machine that supported a global name space. The target machine model of SMARTS is discussed in detail in Chapter 2, where we also discuss architectural support for the operating systems and the RTS's of highly parallel machines. In the next subsection we turn to issues related to parallel and real-time programming languages.

1.1.2 Parallel and Real-Time Programming Languages

Parallel programs consist of sequential *threads* of execution that may be executed concurrently. Cooperating threads may have to communicate information and to synchronize their execution. Many communication and synchronization mechanism have been proposed, and there is no general agreement on which ones are most suitable for any given problem domain. Just as communication and synchronization mechanisms allow threads to control their relative timing, the real-time features of programming languages allow threads to control their absolute timing. Real-time programming languages are even less well understood than parallel programming languages. Before introducing the parallel real-time language Ada, we describe some of the major issues in parallel and real-time programming languages.

Parallel Programming Languages

Parallel programming languages have been surveyed in the literature, for example, in [Andrews and Schneider 1983], [Bal *et al.* 1988] and [Filman and Friedman 1984]. Andrews and Schneider note three central issues in parallel programming languages: How is parallelism expressed? How do parallel threads communicate? And, how do parallel threads synchronize?

Communication allows the execution of one thread to influence the state of another thread. When threads communicate, they may have to synchronize their access to the communication medium. *Synchronization* mechanisms set constraints on the ordering of events. By synchronizing, threads can guarantee that a shared resource (such as a communication mechanism) is accessed by the threads only conditionally or with mutual exclusion. For instance, synchronization mechanisms can be used to prevent two or more threads from racing to access a shared variable. To avoid such *race conditions*, either the execution of some threads must be delayed or the statements within a thread where the variable is accessed must be executed as an indivisible action, i.e., as a *critical section*. Misuse of synchronization mechanisms can wreak havoc. Thus, high-level constructs that force a disciplined use of synchronization mechanisms have been proposed. When a thread blocks or busy waits on an event that will never occur, it *deadlocks*. The absence of deadlocks and livelocks is clearly a desirable feature of parallel programs. A desirable feature of a synchronization mechanism is *fairness*: that a thread waiting to enter a critical section will eventually be able to do so.

Parallel Constructs

Parallelism can be expressed at the statement, subprogram or process level. The parallel execution of a Fortran *do* loop creates threads at the statement level from the iterates of the loop. The C programming language can be extended with the UNIX *fork* statement which creates a parallel thread from a subprogram. (The only way to synchronize with a forked subprogram is to await its completion using a *join* statement.) Processes are best thought of as full blown programs, they have their own stacks, program counters etc. After a process is declared, it executes in parallel with its declarer. Ada *tasks* are defined at the process level.

A language may require that threads be created statically, so that the number of threads can be determined at compile time. Other languages allow threads to be created dynamically at run-time. When the number of threads is allowed to vary, thread termination becomes an issue: a thread must not be destroyed while other threads exist that may need to communicate or synchronize with that thread.

Communication and Synchronization Constructs

Communication between threads is usually accomplished either by *shared variables* (variables accessed by more than one thread) or by *message passing*. There are many synchronization primitives that are based on shared variables, for example, semaphores [Dijkstra 1968] and conditional critical regions [Brinch Hansen 1973] . A semaphore is a non-negative integer variable on which the two atomic operations P and V are defined. $P(S)$ waits until S is greater than zero and then executes $S := S - 1$. $V(S)$ executes $S := S + 1$. (P and V are sometimes referred to as *wait* and *signal*.) Critical sections (where shared resources are accessed) can be implemented using semaphores by surrounding code segments with P and V operations on the same semaphore.

Low-level synchronization mechanisms such as semaphores are easy to misuse. For example, a programmer may omit a V operation (after the corresponding P operation), thereby blocking other potential users of the critical section. Therefore, higher level synchronization mechanism have been proposed[1].

Conditional critical regions (CCR) are a high-level synchronization mechanism based on shared variables. With CCR's shared variables are grouped together into resource classes. (A shared variable can be in only one resource class.) Pieces of code (regions) are prefaced by a resource class R and contain (textually) a condition C; a thread will enter the region only if no other thread is currently executing a region prefaced by R; when the thread reaches the condition C it will be delayed until C is true. CCR's are especially useful for situation where there are *classes* of actors that need access to a shared resource; the actors within a class can access the resource concurrently, but the actors from different classes must exclude each other. An example is the "readers-writer problem" where a variable may be accessed either by Concurrent Readers or an Exclusive Writer (CREW). In this case, the readers would be put in one class and each writer in a class of its own. CREW access can be enforced quite naturally using CCR's (see [Brinch Hansen 1973]). In Chapter 3, where we give implementations of synchronization mechanisms using fetch&add's (the hardware synchronization mechanism of the Ultracomputer and RP3), we discuss a generalization of semaphores and CCR's: *grouplocks* [Dimitrovsky 1988].

When message passing is the means of communication, synchronization is usually achieved with *send* and *receive* primitives. Sends and/or receives may be blocking (synchronous) or non-blocking (asynchronous). They may also be guarded, which means that they are only executed conditionally. Threads may be allowed to wait for multiple sends or receives. However, languages that include constructs for waiting on multiple sends are unusual since these constructs may be quite expensive to implement (it is difficult to retract sent messages). Another important aspect of message passing is how the source and

[1] Many of these high level synchronization mechanism have not actually been included in any programming language, but rather have become programming idioms.

destination of a message are designated. Either direct names, ports or global names may be used. With direct naming thread names are used as the source and destination of messages. Ports allow only one thread to execute a receive on a global name, but any thread may send a message to the port. Global names allow messages to be sent or received by any thread that uses the global name.

Remote procedure calls (RPC's) are a high-level communication mechanism based on message passing. (The procedure is invoked and parameters are passed by messages.) RPC's can be implemented as parallel threads or statements. When each RPC creates a new thread, the RPC is said to be asynchronous. In an asynchronous RPC, the caller does not synchronize with the owner of the remote procedure. On the other hand, when remote procedures are considered statements, the RPC is said to be synchronous, and the body of the remote procedure is executed by its owner. The caller of a remote procedure must block pending the completion of the call. Thus, the body of the remote procedure is executed in mutual exclusion. A synchronous remote procedure may be bound to different bodies at different times (as prescribed by the execution of its owner).

There are three basic parallel programming language models: procedure-oriented, message-oriented and operation-oriented [Andrews and Schneider 1983]. Procedure-oriented languages view the world in terms of objects. There are both active objects (threads) and shared passive objects (e.g., shared variables). Threads communicate and synchronize via shared variables. In message-oriented programming languages, as the name suggests, thread interaction is based on message passing. Threads communicate and synchronize by receiving and sending messages. Operation-oriented languages use the remote procedure call as the basis for thread interaction. The caller and owner threads communicate and synchronize while the operation is being performed.

Procedure-oriented programming languages are most naturally implemented on shared-memory machines and message-oriented programming languages are most naturally implemented on distributed machines. (However, shared variables have been efficiently implemented on distributed machines [Carriero 1987], and message passing can easily be implemented on shared-memory machines [Malony and Reed 1987].) Because operation-oriented languages combine features of both procedure- and message-oriented languages, they are readily implemented on both shared-memory and distributed machines.

Real-time Programming Languages

Real-time applications are inherently concurrent – they must respond to events (stimuli from their environment) that can occur in parallel – and are distinguished by their need to respond to these software and hardware events in real-time. The strictness of these real-time deadlines may vary. For some real-time applications the deadlines are *soft*, and a slightly slower or quicker response time is tolerable. For other applications the deadlines are *hard*, and a too slow or quick response time is disastrous [Faulk and Parnas 1988]. Events can be either *periodic* or *aperiodic*. Periodic events happen at regular intervals, whereas aperiodic events occur only sporadically.

Event Handling

Threads waiting for software generated events can be notified via the thread synchronization/communication mechanism by the thread that generates the event, although other mechanisms specifically for real-time programming have been proposed. For example, Faulk and Parnas [Faulk and Parnas 1988] suggest a synchronization mechanism called *regions states* (based on the classes of actors concept) and a communication mechanism called *state transition events* (operations on a type of shared variable) for real-time applications.

For a thread to be notified by a hardware device of the occurrence of an event, the thread or one of its subprograms must be designated as the *interrupt handler* for that event. The interrupt handler of an event is the code that is to be executed immediately after a hardware interrupt for that event is generated. (Note that the hardware begins executing this code without intervention from the thread scheduler.) Because interrupts must be disabled during its execution, an interrupt handler is often a subprogram that does nothing more than make the thread scheduler aware that the thread whose job it is to handle the interrupt is now schedulable. This newly schedulable thread is called the *device process* or *device driver*. Thus, some facility for specifying device processes and interrupt handlers may be provided by a real-time language.

Specifying Timing

Periodic events happen at predictable times. A thread whose sole job is to handle a periodic event only needs to be executing when that event occurs. Hence, real-time languages may provide some means for suspending a thread until a given time. Real-time programs must also deal with unpredictable environments, where a component may fail, for example. Therefore, threads should be able to *time-out* events. For instance, if a message has not arrived within a given time, the appropriate action might be to cancel the corresponding receive.

Both of the above functions, voluntarily relinquishing a processor until a given time and timing out an event, can be achieved using timer interrupt handlers. However, there is a subtle distinction between how the timer is set for periodic events and for time-outs: A thread that handles a periodic event will want to be awakened at an absolute time, i.e., when the event is known to occur next; a thread that is timing out a rendezvous wants to be awakened at a time relative to when it began waiting for the rendezvous. It is non-trivial to coordinate multiple outstanding timer interrupts and multiple threads designated as the timer interrupt handlers of possibly multiple timers. By including a high-level construct for handling timer interrupts in a language the responsibility of managing multiple timer interrupts is moved from the programmer to the RTS.

Scheduling

When an event occurs that causes a suspended thread to be eligible to run, the thread must be able to regain a processor within a fixed amount of time (so that the event can be processed). This can be accomplished by assigning threads *priorities* and using a *pre-emptive* scheduling discipline: a policy where a thread of higher priority pre-empts the processor of a thread of lower priority. For finer control over the scheduling of threads, the priorities of threads may need to vary dynamically.

A pre-emptive scheduling policy, however, is not sufficient to guarantee that a high priority thread will meet a deadline. For instance, pre-emptive scheduling does not rule out priority inversion, a situation where a thread is blocked by a thread of lower priority. Priority inversion can occur, for example, when a medium priority thread M pre-empts the processor of a lower priority thread L while there is a higher priority thread H waiting for an event from L. A (partial) solution to priority inversion is *priority inheritance*: a thread executes at the maximum of its own priority and that of any other thread that is waiting to be unblocked by the thread. While language (and architectural) support can reduce the potential for priority inversion, they are not enough to eliminate it. However, when these measures are taken in conjunction with a disciplined programming style the duration of time that priorities are inverted can be bounded [Goodenough and Sha 1988]. (A discussion of priority inversion is beyond the scope of this thesis. Interested readers are referred to [Goodenough and Sha 1988] and its attendant references.)

Because implementing pre-emptive scheduling can be costly (especially on highly parallel or widely distributed machines), a *prioritized run-until-blocked* scheduling discipline may be preferable for soft real-time applications (applications with soft deadlines). Under a prioritized, run-until-blocked scheduling discipline, a thread runs on a processor until it blocks (i.e., it must wait for an event), at which point the processor begins executing the thread that is eligible to run with the highest priority. To reduce the cost of scheduling threads even further, a pre-scheduling policy, where the assignment of threads to processors is made at compile-time, may be necessary [Faulk and Parnas 1988]. As can be seen, real-time applications may require a wide range of scheduling disciplines. It has been argued that real-time languages should, therefore, include very low-level scheduling primitives upon which other real-time scheduling algorithms can be built [Sha *et al.* 1987].

Finally, a high-level real-time programming language may provide features to allow a program to have low-level control over data or code so that timing constraints can be met. For instance, to ensure that a time critical piece of code meets a strict deadline, it may be necessary to specify both the data layout and the actual machine code that is executed.

1.2 The Ada Tasking Model

Many of the parallel and real-time features that we have just described are included in the programming language Ada. Ada is a high-level block structured, operation-oriented language that was designed to write large-scale and real-time systems. To aid in large programming efforts, the design of Ada stresses the reliability and maintenance of Ada programs. For instance, Ada includes support for reusable software libraries and has a built-in exception handling mechanism. Since real-time programs are inherently concurrent, Ada includes both parallel and real-time features.

Concurrency in Ada is expressed at the process level using *tasks*. Task communication and synchronization is accomplished in an architecture independent, high-level manner with synchronous remote procedure calls, called *rendezvous*. (Asynchronous remote procedure calls, i.e., shared subprograms, and shared variables are also supported.) Ada's real-time features include device processes, interrupt handlers, delays and pre-emptive scheduling. They permit tasks to control their timing and hardware events. The tasking model of Ada is defined in Chapter 9 of the Ada Reference Manual (ARM) [DoD 1983], we give an overview of it here. The suitability of the Ada tasking model for writing application programs for highly parallel machines is discussed in §1.3.

The following example illustrates the main constructs that comprise the Ada tasking model. We will refer to this example in the next two subsections where we describe Ada's parallel and real-time features (respectively).

Example 1. Ada Tasking

```
-- A sever task to perform synchronized operations on integer variables in global memory.
-- See Chapter 5 for an in-depth discussion of this example which is an adaptation of
-- the beacon tasks presented in [Schonberg and Schonberg 1985].

    -- Type of the address (location) of an integer in global memory.
    -- Note that the selector all is used to denote (the value of) the integer.
    type SHARED_INTEGER is access integer;

    -- Global memory integer synchronizer task specification, i.e., the interface
    -- it presents to other program units.
    task type integer_synchronizer is
        -- Entry for reading a shared variable.
        entry read(I : out integer; V : in SHARED_INTEGER);
        -- Entry for writing a variable in global memory.
        entry write(V : in out SHARED_INTEGER; E : in integer);
        -- Entry for performing, I := fetch&add(V, E).
        entry fetch_and_add(I : out integer; V : in out SHARED_INTEGER; E : in integer);
        -- Entry for performing, I := fetch&store(V, E).
        entry fetch_and_store(I : out integer; V : in out SHARED_INTEGER; E : in integer);
    end;

    -- A task object with an anonymous type.
    task client;
     -- A task object of type integer_synchronizer.
    integer_memory_monitor : integer_synchronizer;

    -- The body of task client, i.e., its implementation.
    task body client is
    I : integer;
    V : SHARED_INTEGER := new integer;
    begin
        integer_memory_monitor.write(V, 0); -- Initialize V.all.
        select -- Select containing an entry call and a delay alternative.
        -- Increment V.all by 2, if the rendezvous takes place before the delay expires.
            integer_memory_monitor.fetch_and_add(I, V, 2);
        or
            delay 0.5;                          -- If we can't rendezvous in time then...
            abort integer_memory_monitor.     abnormally terminate integer_memory_monitor.
        end select;
end client;

-- Global memory integer synchronizer task body, i.e., its implementation.
task body integer_synchronizer is
begin
    loop
        select  -- Select containing accept statements and a terminate alternative.
            -- V.all := E;
            accept write(V : in out SHARED_INTEGER; E : in integer) do
                V.all := E;
            end write;
        or -- I := V.all;
            accept read(I : out integer; V : in SHARED_INTEGER) do
                I := V.all;
            end read;
        or -- I := fetch&add(V, E);
            accept fetch_and_add(I : out integer; V : in out SHARED_INTEGER;
                                 E : in integer) do
                I := V.all;
                V.all := V.all + E;
            end fetch_and_add;
        or -- I := fetch&store(V, E);
            accept fetch_and_store(I : out integer; V : in out SHARED_INTEGER;
                                   E : in integer) do
                I := V.all;
                V.all := E;
            end fetch_and_store;
        or -- Terminate when there are no more customers.
            terminate;
        end select;
    end loop;
end integer_synchronizer;
```

1.2.1 Parallel Features of Ada

Ada tasks are first class objects – every task object has a (possibly anonymous) task type; aggregates (such as arrays) can be composed of tasks; there can be pointers to tasks (*access tasks)* and tasks can be passed as parameters (with some restrictions). (In example 1, the task *integer__memory__monitor* is of type *integer__synchronizer,* while the task *client* has an anonymous type.) Tasks are created dynamically at run-time. When a task creates a new task, the execution of both tasks proceeds in parallel. The initial phase of execution of a task during which it elaborates its declarations is called the *activation* of the task. (Although Ada tasks are defined at the process level, .i.e, as virtual programs, a task cannot be passed parameters when it is activated.)

Tasks are visible to other tasks in accordance with the usual scope rules of block structured languages. Hence, a task object cannot be destroyed until after the block (task, subprogram or block) that declares the task object is left, and a task pointer cannot be destroyed until after the block that declares the task pointer type definition is left (this ensures that there are no dangling references). A task object is said to be a *dependent* of the task that declares the task object; a task pointer is a *dependent* of the task that declares the task pointer type definition. A task that has dependent tasks is said to be the *master* of these dependents.

To facilitate the termination of server tasks, tasks that provide services to other tasks via rendezvous, Ada provides the *terminate* statement. (The body of a server task type, *integer__synchronizer*, is depicted in example 1.) The terminate statement allows sets of tasks that are related by dependency relationships to request to be terminated together when they all become idle (where an idle task is one that has either completed its execution or is waiting to provide a service). (Terminate statements are one of the possible alternatives to the select statement described below.)

Tasks can also explicitly *abort* (cause to be abnormally terminated) other tasks. A task that is suspended waiting for an event, such as a rendezvous, when it is aborted must also be unblocked. All of the tasks that depend on the aborted task must be aborted. When a task attempts to rendezvous with a task that has finished its execution or been aborted, a predefined exception TASKING__ERROR is raised.

The usual means for two tasks to communicate and synchronize is a rendezvous. A rendezvous takes place when one task (the *caller*) calls an *entry* of another task (the *owner*), and the owner reaches an *accept* statement for that entry. At this time the statements associated with the accept are executed, after which both of the tasks resume execution. The owner may have more than one body (accept statement) for an entry. (I.e., rendezvous are synchronous RPC's which can be bound to different bodies.) Entry names are ports; therefore, any task that can see an entry may call that entry, but only the owner may accept calls to the entry. If there are no tasks waiting at an entry the accepting task blocks. If an entry call cannot be accepted the caller is queued (until either the entry call is accepted or it is canceled). The tasks queued at an entry will be accepted in the order of their arrival.

In example 1, the services offered by the server task *integer_memory_synchronizer* (which is of type *integer_memory_monitor*) are its entries: *read, write, fetch_and_add* and *fetch_and_store.* Other tasks, such as *client,* request these services by executing entry calls.

S*elect* statements allow a task to wait on the accept statements of multiple entries. These accept statements may be conditional, i.e., guarded. (Select statements are not guaranteed to be fair: It is not defined by the language which of the open accept alternatives of a select statement will be chosen for a rendezvous.) Select statements may optionally have alternatives that are taken if the rendezvous cannot occur within a certain time. The alternative is chosen if the rendezvous cannot begin immediately in the case of an *else* alternative, in a specified amount of time in the case of a *delay* alternative (delay statements are discussed below in the real-time features section) and indefinitely in the case of a terminate alternative. While a task can have only a single outstanding entry call, select statements that contain an entry call and either an else or a delay alternative can be used to time-out entry calls.

Returning to example 1, the task *client* has specified that it is not willing to wait forever to rendezvous with *integer_memory_monitor* by enclosing its entry call in a select with a delay alternative. If the delay expires before the rendezvous has begun, then *client* will abort (abnormally terminate) *integer_memory_monitor.* By enclosing its accept statements in a select statement that has a terminate alternative, *integer_memory_monitor* has requested to be terminated when there are no tasks left to call its entries.

A task is allowed to reference subprograms and variables local to other (enclosing) tasks. These shared subprograms and variables are an alternative way for tasks to communicate and synchronize. Access to shared variables is restricted: Between points where tasks explicitly synchronize (e.g., rendezvous) shared variables must be accessed either read-only or by a sole writer (CREW). Hence, tasks are allowed to reference local copies of shared variables between points where they synchronize (which can dramatically reduce the access times of shared variables on multiprocessors with local caches or memories). The Ada language also provides a *pragma SHARED* which causes every access (read or write) to a shared variable to be a synchronization point for that variable, which is equivalent to saying that no local copies of variables named in a pragma SHARED can be kept. The pragma SHARED can only be applied to variables whose type is scalar or access (This type of variable can be updated atomically on many machines.).

1.2.2 Real-time Features of Ada

As mentioned above, the features of Ada that allow a task to respond to hardware events and to control its timing are: device processes, interrupt handlers, delays and pre-emptive scheduling. By associate the address of the device with one of its entries, a task can be designated as the process (or driver) for that device. Entries that are associated with the address of a device are referred to as *interrupt entries*. Accept statements for interrupt entries are "called" when the device issues a hardware interrupt. Thus, the entry call acts as the interrupt handler for the device. The ARM states that interrupt entry calls can be implemented by having the hardware execute the accept statement directly When executed directly by the hardware, accept statements for interrupt entries will themselves be part of the interrupt handler

The *delay* statement of Ada suspends the execution of a task for at least the specified duration. Recall that, a delay that is an alternative of a select statement times out the corresponding entry call or accept statement (see the task *client* in example 1).

Ada tasks may specify a static *priority*. When tasks specify priorities, Ada requires that a pre-emptive scheduling policy be used. As a limited form of priority inheritance, a rendezvous between two tasks that have specified priorities is executes with the maximum of the priorities of the tasks. If only one of the tasks engaging in a rendezvous has a specified a priority, then the rendezvous is executed with at least that priority. (Hardware interrupt entries execute at a priority that is higher than the highest priority that can be specified for a task.) Similarly, a task is activated at the maximum of the priorities of the task and its parent.

Representation clauses allow a programmer to control the layout of data. Machine code can be inserted into an Ada program using the predefined library package MACHINE_CODE and the pragma INLINE. Ada also provides a pragma INTERFACE which allows procedures written in other languages can be called from Ada programs.

The parallel and real-time features of Ada are discussed in more detail in Chapter 4, where we describe how they are implemented in SMARTS. In §1.4 we consider some of the difficulties in implementing RTS's for parallel real-time languages such as Ada. The suitability of Ada for writing code for highly parallel machines is the subject of the following section.

1.3 The Suitability of Ada for Highly Parallel Machines

Proposed highly parallel shared-memory MIMD machines such as the NYU Ultracomputer and IBM RP3 have traditionally been regarded as suitable for large-scale scientific code, and not suitable for more symbolic or heterogeneous concurrent applications, such as are found in artificial intelligence or real-time computing systems. However, these applications would benefit greatly from (and even require) the computational power provided by parallel machines ([Haynes *et al.* 1982], [Middlemas 1987] and [Stankovic 1988]).

The intended application domain of Ada is large systems and real-time programming. Although the Ada tasking model was primarily designed to write programs for controlling embedded systems, because of Ada's rich set of tasking features and support for programming in the large, computationally-intensive numeric and symbolic applications are increasingly being written in Ada

Scientific problems, for instance, have been expressed naturally using Ada tasks (e.g., [Hibbard *et al.* 1981] and [Schonberg and Schonberg 1985]) and scientific applications are being coded in Ada (e.g., [Klumpp 1987] and [Sebesta 1987]). Ada's emphasis on program reliability and maintenance facilitates the (static) verification of large numerical programs.

Ada is also being used to implement artificial intelligence applications and computer simulations (e.g., [Melde and Gage 1987] and [Schultz and Chandna 1987]). Artificial intelligence and simulations have commonly been the domain of languages such as Smalltalk or LISP. However, in two recent studies ([Walters 1987], [Seidewitz 1987]), Ada, because of its life-cycle software engineering orientation, compared favorably to both LISP and Smalltalk for these types of applications[3].

The most common tasking idiom for scientific code written for highly parallel machines describes a large number of nearly identical short-lived tasks (declared as an array of tasks in Ada) operating on a shared data structure (such as a matrix). This paradigm will fit some artificial intelligence applications (such as low to intermediate level vision) as well. Other artificial intelligence applications will be characterized by longer-term, perhaps more

[2] The only area where Ada was found to be inferior to Smalltalk was its lack of support for inheritance. Interestingly enough, it is the inheritance mechanism of Smalltalk that has been criticized as not being well suited to multiprocessor environments [Bennett 1987]

heterogeneous, tasks [Hummel and Zhang 1987] (again, operating on a large shared data structure, e.g., agents communicating via a blackboard). Real-time applications for the next generation of parallel machines are also apt to consist of longer-term, more heterogeneous tasks ([Stankovic 1988]). For efficiency reasons real-time applications may well rely on shared variables for synchronization and communication ([Faulk and Parnas 1988]). An Ada compiler must, therefore, support not only large-scale parallelism, but a wide variety of programming styles if it is to realize the potential of highly parallel machines.

Ada compilers for various architectures have been or are being implemented: For uniprocessors many efficient compilers and RTS's have been written (e.g., Alsys, DEC, [Riccardi and Baker 1984] and [Rosen 1983]). Several Ada compilers exist for tightly-coupled multiprocessors with less than 50 processors (e.g., Alliant, [Ardö 1984], [Newton 1987] and Sequent). There are research projects underway for implementing Ada on distributed machines (e.g., [Jha and Kafura 1985] and [Fisher and Weatherly 1986]).

Given that efficient Ada compilers currently only exist for multiprocessors with a modest number (less than 50) of processors[4], the question remains: Will compilers be efficient enough to allow applications written in Ada to effectively harness the power of large numbers of processors? It is the tenet of this thesis that Ada's parallel and real-time features can be implemented efficiently, even on highly parallel machines.

In the rest of this section we address criticisms that have been made in the literature regarding the suitability of Ada for writing application programs for highly parallel

[3] The Ada tasking implementation described in [Newton 1987] is targeted to single and four processor VAX's running the MACH operating system [Tevernian and Rashid 1987]. His implementation could potentially be ported to highly-parallel shared memory machines such as the Butterfly GP 1000 parallel processor (which has up to 256 processors), since the GP 1000 runs a version of MACH (MACH 1000) [Russel and Waterman 1987]. While this certainly would be an improvement over the current Ada compiler available for the GP 1000 (which only runs on a single processor), we do not expect this solution to effectively scale upward. For example, preliminary tests by Newton of the implementation on single and four processor VAX's showed that a limit is reached (between 10 and 50 tasks) after which any new task created incurred considerable additional expense.

machines. We argue that concerns over the suitability (or expressiblity) of the parallel and real-time features of Ada are, for the most part, misplaced. It is not the expressiblity of Ada that is in question, it is whether Ada's tasking model can be efficiently implemented that has been suspect. (Admittedly, it is difficult to divorce expressiblity issues from efficiency issues of parallel real-time languages – the very purpose of executing programs in parallel is to reduce their execution times and real-time applications must, by definition, deal with time.) We show that most of the deficiencies of Ada discussed below can be corrected by compile-time techniques. In §1.4 we considers the functions that must be efficiently supported by an Ada compiler at run-time.

1.3.1 A Critique of Ada's Tasking Model

The appreciation of Ada is by no means universal. Indeed, over the years Ada has been roundly criticized. The language has been declared too complex ([Hoare 1981] and [Ledgard and Singer 1982]). The parallel features of Ada have been deemed as being unsuitable for expressing certain classes of parallel algorithms ([Yemini 1982] and [Shulman 1987]). Recently, the real-time features have come under attack as being both too weak and too high-level to be efficiently implemented [Brosgol 1988]. We address some of these criticisms here.

Since we are mainly concerned with highly parallel multiprocessors, we restrict our discussion to the parallel and real-time feature of Ada as they pertain to this class of machines. We justify this by noting that efficient Ada compilers for uniprocessors and multiprocessors with a small number of processors exist. Hence, we can conclude that the sequential features of Ada are not too cumbersome to admit efficient implementations, and further, that Ada's tasking model can be supported efficiently on a modest number of processors[4]

This subsection is not meant to be a defense of every feature of Ada, the language is not perfect. We only wish to demonstrate that, given an efficient implementation with the proper compile and run-time support, Ada is a more than reasonable language for programming highly parallel machines.

Pragmas and Library Packages

Before discussing specific criticisms of Ada, we present two general strategies that have been used to overcome the shortcomings of Ada: implementation defined pragmas and specialized packages. We also introduce some techniques, translation schemes, for efficiently implementing the features of Ada which have been criticized as being intrinsically inefficient. These translation schemes can be either automatic or programmer assisted. Several of the pragmas, specialized packages and translations schemes we outline here are supported by SMARTS (see Chapters 4 and 5 for details).

The pragma and library package facilities of Ada can be used to tailor the language to a particular application or environment. For instance, implemenation-defined packages can be provided for features that are supported efficiently on a given architectures. (Programs which rely on implementation-defined library packages and pragmas will obviously not be portable.)

An Ada implementation can be further integrated into its environment through the predefined pragma INTERFACE and the predefined library package MACHINE CODE. The pragma INTERFACE permits Ada programs to include calls to procedures written in other languages, while the predefined library package MACHINE CODE permits Ada programs to contain machine code inserts. It is good programming practice to place calls to programs

[4] It is interesting to chronical the progression of the criticisms that have been made of Ada: First, the language was criticized as being too large to be implemented efficiently on uniprocessors. Efficient uniprocessor implementations followed. Then, Ada's parallel features were criticized as being too complex to be efficiently supported on multiprocessors. Efficient multiprocessor implementations followed. Now Ada's temporal features are being criticized as being unsuitable for real-time applications. Next?

written in other languages and machine code inserts in library packages, so that only these packages will need to be rewritten when the program is ported to a new machine. The Ada Runtime Environment Working Group has suggested that Ada implementations provide such packages, consisting of low-level task scheduling primitives [ARTEWG 1986].

Packages that are known to the compiler can also be used to optimize higher-level tasking idioms: Many of the perceived inadequacies of Ada for writing code for highly parallel machines can be overcome by providing a library of tasking idioms that will be executed efficiently on the target architecture. Another possibility is to have the compiler recognize these idioms automatically and translate them into efficient code. A simple automatic translation optimization is for a compiler to recognize when programs do not use expensive features, such as abort statements, and hence, be able to generate more efficient code. (The mere possibility of tasks being aborted increases the cost of implementing almost all of Ada's tasking features – see §4.7.) Until Ada compilers become more mature, however, it may be necessary for programmers to assist the compiler in performing such optimizations by adhering to a disciplined programming style.

A Critique of Ada's Parallel Features

The main criticisms of Ada's parallel features have centered around its synchronization and communication mechanisms, rendezvous and shared variables.

Rendezvous

There are two perceived drawbacks to using rendezvous in highly parallel code: they are too high-level and they are binary. Because rendezvous are complex they may be fairly expensive to implement (in one implementation rendezvous were reported to be an order of magnitude more expensive then procedure calls [Burger and Nielson 1987]). If tasks rely on costly rendezvous for communication and synchronization, then the smallest granularity of task that it is worth executing in parallel may be unduly limited. To make effective use of machines consisting of hundreds or thousands of processor fine grain parallelism (on the order of a few instructions [Gottlieb 1984]) may be necessary. Moreover, since rendezvous with client tasks must be executed serially by a server task, it has been assumed that such rendezvous idioms will cause a bottleneck when used to coordinate large numbers of tasks. Worse yet, in some cases (e.g., [Newton 1987]) the overhead due to contention from client tasks attempting to rendezvous with a server task was found to increase superlinearly with the number of tasks involved.

Other aspects of the Ada tasking model are impacted by an inefficient implementation of rendezvous. For instance, Ada tasks cannot be passed parameters when they are activated, and therefore, must be passed this information via rendezvous. In particular, rendezvous must be used to inform a task of its own identity (for example, a task that is an element of an array does not know which element it is unless it is explicitly told).

Both implementation-defined pragmas and a package of low-level tasking primitives are possible ways of circumventing the inefficiencies associated with rendezvous. Automatic translation schemes are more satisfactory, since these do not adversely affect the portability of programs. Another advantage that translation schemes have over packages is that the programmer can reason about the program using high-level constructs (without sacrificing efficiency). Accordingly, many strategies for efficiently implementing certain tasking idioms involving rendezvous have been proposed, e.g., [Habermann and Nassi 1980], [Hilfinger 1982], [Lindquist and Joyce 1985], and [Stevenson 1980]. [Schonberg and Schonberg 1985] extend this concept to cover highly parallel machines. They define a tasking idiom, the beacon task, whose rendezvous can be transformed on certain architectures (e.g., the Ultracomputer and RP3) into efficient operations on shared variables. Using beacon tasks a large number of tasks can be initialized in a bottleneck-free fashion. In Chapter 5, we discuss tasking idioms that should be included in a library of an Ada compiler for a highly parallel machine.

Shared Variables

Shared variables, the alternative to rendezvous for task communication and synchronization in Ada, have also been criticized. The lack of a cohesive storage model for Ada shared variables and the restrictions that Ada places on what variables can be named in a pragma SHARED (only variables that are of scalar or access type) cause many useful parallel programs that rely on shared variables to be classified as erroneous when written in Ada (for details see §4.8 and [Shulman 1987]). For example, recall that the expected programming paradigm for large-scale scientific code is a large number tasks operating on a shared composite object. The inability to name such shared composite objects in a pragma SHARED seemingly outlaws this style of programming. Ada forces task to access shared composite objects via rendezvous even when synchronization is not required.

While some of the deficiencies of shared variables pointed out by Shulman are more serious (and may even necessitate a revision of the language), the problem with shared composite objects can be overcome using pragmas or translation techniques. A package of functions that allow the hardware to perform operations on arrays of data in parallel (i.e., vector operations) is presented in [Völksen and Wehrum 1986]. A scheme is described in Chapter 5 for optimizing away rendezvous whose sole purpose is to allow tasks to access a shared composite object. The DEC Ada compiler supports a pragma VOLATILE. Like the predefined pragma SHARED, local copies of variables named in a pragma VOLATILE cannot be kept. However, unlike the pragma SHARED, variables named in a pragma VOLATILE are not restricted to those of scalar or access type.

A Critique of Ada's Real-time Features

The most often maligned real time features of Ada have been its delay statement and scheduling policy

Delay Statement

The Ada reference manual states that a delay must cause a task to be suspended for *at least* its duration (which seems reasonable). Many have claimed that it would be more useful to have the language specify the maximum that a task can be delayed rather than the minimum (see for example, [Brogsol 1988]).

Specifically, it has been argued that the points at which a delay must have expired (i.e., delay synchronization points akin to abort and shared variable synchronization points, see §4.6 and §4.7) should be precisely defined by the language. However, if such delay synchronization points were defined in terms of the execution of another (non-suspended) task, then they could not be tested – the delayed task is suspended, and hence, cannot itself participate in the synchronization point. Alternatively, if delay synchronization points were defined in terms of elapsed time (measured in instructions or seconds) then either this time would have to be made arbitrarily large or else many architectures, especially distributed machines, would be excluded from running Ada by virtue of never being able to meet the required deadline. Clearly, the maximum time to expire a delay is an implementation-dependent feature. Ada does not *require* compilers to implement delays inefficiently, and market forces dictate that they implement them as efficiently as possible, i.e., by scheduling a task as soon as possible after its delay has expired.

A more serious criticism of Ada's delay is that only a relative time can be specified. Tasks that must handle periodic events need to awoken at absolute times. While a task could calculate the time remaining on its interval and then execute a delay for this duration, this solution is not satisfactory since the task may be pre-empted before it can execute the delay. Alternatively, a compiler could recognize such sequences (of calculating the time remaining on an interval and then executing a delay) and translate them into a delay for the appropriate duration. Another possibility is to include a pragma ALARM which would delay a task until a specified time.

Scheduling Policy

The pre-emptive scheduling policy of Ada has been criticized for not specifying when a newly executable task must pre-empt the processor of a lower priority task. This is similar to the criticisms made about the maximum time that a task must be suspended when it executes a delay. Pre-emptive scheduling synchronization points (the points at which a newly executable task must pre-empt the processor of an already running task of lower priority) have the same problems as delay synchronization points: the newly executable task cannot itself participate in the synchronization point, and to encompass all architectures, synchronization points defined in terms of elapsed time would have to be made arbitrarily large. Again, the maximum time for a newly executable task must pre-empt the processor of an already running task of lower priority is an implementation-dependent feature, and hence, not something that should be defined a priori by the language.

Other, legitimate, complaints have been made about Ada's scheduling mechanism. Ada does not allow programmers to have enough control over the scheduling of tasks. The scheduling requirements of every real-time application do not fall at one of the two extremes defined in the reference manual: no priorities or pre-emptive. Further, the priority inversion problem is exacerbated by: static priorities, the FIFO (rather than priority based) acceptance of tasks on entry queues, and the non-deterministic (again, rather than priority based) selection of the open alternatives of select statements.

The FIFO acceptance of entry calls appears to be a more serious impediment to the avoidance of priority inversion than the non-deterministic selection of select alternatives, since an implementation is not even free to choose to accept tasks in priority order. Fortunately, *entry families* (arrays of entries) can be used to circumvent the FIFO acceptance of entry calls by having one element in the entry family per priority level (see [Burns 1987] for details).

It has also been claimed that even executing a rendezvous at the maximum priority of the two task involved does not go far enough in avoiding priority inversion – it does not take into account the priorities of the other tasks that are waiting on the entry queue. However, when the priority of one of task participating in a rendezvous is undefined, the rendezvous is only required to be executed with *at least* the priority of the other task (i.e., executing the rendezvous at some higher priority is legal). Thus, by not specifying the priority of a server task, an implementation is free to execute the rendezvous of the task at the highest priority of a task on its entry queues [Dewar 1988a].

As mentioned earlier, priority inversion is not unique to Ada and can only be solved by using a disciplined (or restricted) programming style. One such programming paradigm that minimizes priority inversion is the ceiling protocol which places restrictions on the usage of accept statements and entry calls. The ceiling protocol has been successfully followed in the design of a real-time system written in Ada [Locke and Goodenough 1988]. Nevertheless, an Ada compiler could contribute to the elimination of priority inversion and satisfy the widely varied scheduling requirements of real time applications by providing the appropriate packages and pragmas. For example, a prioritized run-until-blocked scheduling policy based on priorities declared with implementation-defined pragmas could be supplied by an RTS. A package of low-level tasking primitives which allow applications to do their own task scheduling has been implemented by Baker and is discussed in [Baker 1987].

Whether the solutions we have sketched are adequate to solve the deficiencies of Ada's real-time features is not clear. More experience with actual implementations is needed. It has been suggested that Ada is only suitable for soft real-time applications and not hard [Brosgol 1988]. However, the jury is still out. In fairness to Ada, no other high-level language has support for as many real-time features. Indeed, it has been argued that these problems are not unique to Ada, and that by adhering to the real-time programming styles that have proven successful for other high-level languages, Ada will be more than adequate for writing hard real-time applications such as avionics [Roark and McAfee 1988].

1.4 A Run-Time Supervisor for Ada on Parallel Machines

A RTS for a parallel real-time language on a multiprocessor must provide run-time support for the parallel and real-time features of the language. When the number of processors, and hence the number of tasks managed by the RTS, is large, it becomes imperative that the RTS not contain serial bottlenecks. (In the remainder of this thesis, we will use Ada terminology, e.g., *task* instead of *thread*.) Therefore, the RTS functions themselves must be implemented in a highly parallel distributed manner.

To be able to respond to events in real-time, the RTS must ensure that a task can pre-empt the processor of a currently executing task within a guaranteed amount of time. A consequence of this requirement is that hardware interrupts can only be masked out for short periods of times. Accordingly, RTS functions must implemented with only extremely short critical sections. (Critical sections that provide mutually exclusive access to the state information of tasks are often required to avoid race conditions that would otherwise arise from the parallel implementation of RTS functions.)

Thus, ideally, RTS's for parallel real-time languages on highly parallel machines should be efficient, highly parallel and critical section free. Currently, there is no consensus on how these goals should be achieved In this section we present some of the functions that must be performed by an Ada RTS, and some of the inherent difficulties in implementing these functions. (Note that an Ada RTS must also provide run-time support for its implementation defined pragmas and packages. We do not consider the features provided by these pragmas and packages here, because they are obviously implementation-dependent In Chapters 4 and 5 we do discuss the implementation of the pragmas and packages with which we augment SMARTS.) The basic functions that must be provided by an Ada RTS in a multiprocessor environment are listed below. We also identify some of the hindrances to their efficient, highly parallel and critical section free implementation. See Chapter 4 for details on how SMARTS manages to circumvent these hindrances.

Task Creation and Activation

Ada tasks are created, and hence, activated dynamically at run-time. If tasks must be created or activated serially, then a bottleneck will ensue when Ada programs use large arrays of tasks (the expected tasking idiom for scientific code written for highly parallel machines).

Task Termination and Destruction

An Ada RTS must detect when sets of tasks related by dependency relationships are ready to terminate, i.e., when they have completed their execution or are waiting at a terminate alternative. Detecting when sets of tasks can terminate, and releasing their storage are potentially serial bottlenecks. Worse yet, the RTS seemingly must decide if all of the tasks in the set containing a task agree to terminate before accepting an entry call for that task.

The abort statement of Ada causes a task to be terminated prematurely. The implementation of an abort statement is another potential serialization point, since all of the dependents of an aborted task must also be aborted.

Scheduling

Some mechanism for dynamically scheduling tasks on processors (such as a queue of ready-to-run tasks) must be included in an Ada RTS. Since tasks may have to be scheduled on many processors simultaneously, we must ensure that task scheduling can be accomplished in a highly parallel manner.

Scheduling is further complicated by priorities – tasks of higher priority must be scheduled before tasks of lower priorities. Further, since Ada requires a pre-emptive scheduling policy, when a task of priority *P* becomes executable and there are no idle processors, it is necessary to determine if any task of priority less than *P* is currently executing so that its processor can be pre-empted. On a highly parallel multiprocessor, determining the task of lowest priority may take a non-negligible amount of time. Furthermore, many tasks may simultaneously become executable. Each of these task may have to pre-empt the processor of a currently executing task. Thus, the pre-emption of processors is another RTS function that should be free of serialization points. For real-time applications the pre-emption of a processor must be completed within a guaranteed amount of time. So that this period of time is as small as possible, the length of the critical sections contained in the RTS during which a task cannot be pre-empted must be minimized.

Finally, while Ada does not require that a compiler detect deadlock, the RTS must detect when all the tasks are irrevocably blocked in order to terminate the program promptly.

Time Management

Delays serve two purposes. They can be used to suspend a task for given duration and to time-out rendezvous. A multiprocessor may have a single global clock, a clock per processor, or both. Using a single timer interrupt handler for the global clock is potentially a serial bottleneck. If there is a timer interrupt handler for the clock of each processor, then it must be possible for a task executing on one processor to disable a time-out that was set on a different processor. (A time-out for a task must be disabled if a rendezvous occurs or if the task is aborted.) The disabling of a time-out for a task may have to be executed as a critical section.

Communication and Synchronization

Ada's has a rich set of communication and synchronization mechanisms that must be supported: rendezvous (synchronous RPC's), shared subprograms (asynchronous RPC's) and shared variables. Although the expressive power of rendezvous make them well suited for writing parallel programs, it also makes them difficult to implement efficiently. Part of this difficulty arises from the fact that tasks are able to nest rendezvous and to wait on multiple accept statements. The implementation of rendezvous is further complicated by its interaction with other Ada constructs, most notably, terminate statements, aborts, delays and shared variables. Terminate alternatives complicate the acceptance of entry calls. To implement aborts and delays an RTS must be able to cancel accept statements or entry calls. The presence of shared variables mandate that accept statements to be executed in the environment of the caller. Finally, although a rendezvous takes place between two tasks, the callers of an entry that cannot be accepted immediately must be queued; enqueuing the callers of an entry may be a bottleneck.

Schemes have been proposed for implementing rendezvous with only one task context switch: whichever task arrives last (the caller or the owner) executes the rendezvous ([Habermann and Nassi 1980], [Lindquist and Joyce 1985] and [Stevenson 1980]); however, there is a large overhead associated with these schemes, due primarily to nested rendezvous, aborts and shared variables. Even if this translation scheme was viable, rendezvous would still be binary, that is to say, they only allow tasks to synchronize two at time. Such pair-wise synchronization of tasks will be a serialization point. Clearly, more sophisticated rendezvous translation schemes are needed if rendezvous are to be used to effectively coordinate large numbers of Ada tasks.

The semantics of shared subprograms is much simpler than that of rendezvous. Like rendezvous, however, they must be executed in the environment of the owner of the subprogram. (As described below, having to execute a rendezvous and shared subprograms in the environment of the owner make the management of storage more difficult.)

The restrictions that Ada places on how shared variables that are not named in a pragma SHARED are accessed and what shared variables can be named in a pragma SHARED are meant to simplify their run-time management on multiprocessors with a hierarchy of memories (e.g., cache, local and global). For instance, they permit a task to keep local copies of shared variables that are not named in a pragma SHARED between synchronization points. However, to take full advantage of this optimization we must know precisely what variables are shared (see §4.8 for details). Unfortunately, only Ada shared variables that are named in a pragma SHARED are easily identifiable. The compile-time detection of shared variables that are not named in a pragma SHARED may be difficult or impossible.

Storage Management

The management of storage for parallel real-time languages in a multiprocessor environment is complicated by the presence of several active task and possibly more than one memory (i.e., local memories and a global memory). Block structured languages are usually implemented with a stack of activation records (one record for each currently invoked subprogram). Non-local references are resolved using displays or static links. These data structures do not extend naturally to parallel languages.

The parallel language variant of an activation stack is a *cactus* stack. The activation stack of each task creates a new branch. Cactus stacks (as they have traditionally been implemented) may be problematic for multiprocessors for several reasons. First and foremost of these is that there is no clear demarcation or separation of shared and non-shared (private) data within an activation record. On a hybrid architecture, shared data must be allocated in the global memory and private data in the local memories. (Since accessing variables stored in non-local memories via displays or static links may be expensive, care must be taken when deciding where to store data.) Furthermore, since stacks grow and shrink, the same location in an activation record may contain different types of data at different times. Compounding this data placement problem is that fact that not all Ada shared variables are easily identifiable – a variable becomes shared simply by virtue of being accessed by more than one task. Thus, to implement shared variables efficiently in a multiprocessor environment, new techniques and data structures are required.

Cactus stacks also do not lend themselves to efficient implementations of RPC's. The shared variables referenced within accept statements and shared subprograms (i.e., the environments of the accept statement or subprogram) are not easily discernable, and hence captured, from a cactus stack (these variables will be scattered throughout the activation records of the ancestors of their owner).

Many of the RTS functions that we have listed in this section are typically provided by operating systems. In general, RTS for real-time, parallel programming languages on multiprocessors bear strong resemblance to operating systems for multiprocessors. The comments made in §2.1.1 pertaining to desirable features of operating systems for highly parallel machines apply equally as well to RTS.

1.5 Present Work: SMARTS

In the two preceding sections, we considered the suitability of Ada for writing applications to be run on highly parallel machines and we enumerated the functions that must be efficiently supported by an Ada RTS for highly parallel machines. Ada was found to be well suited to writing such applications because of its rich tasking model and support for programming in the large. The concern has been that the Ada RTS's functions cannot be efficiently implemented on large-scale multiprocessors. That is, the cost of managing the large number of Ada tasks necessary to make effective use of highly parallel machines will negate the speedup which is the very reason for their parallel execution.

We note three main arguments against the use of Ada on highly parallel shared-memory machines. First, Ada tasks have typically been implemented as relatively high-level or *heavy-weight* processes, and therefore, unsuitable for fine-grain parallel applications. Second, rendezvous are inappropriate for coordinating large numbers of tasks, and as currently defined by ARM, so are shared variables. Third, and most importantly, an RTS for

Ada must contend with many potentially expensive serialization points and functions that may require mutually exclusive access to task state information. For parallel architectures with hundreds or thousands of processors, where potential gains from parallelization are substantial, it is imperative to avoid serial bottlenecks and minimize critical sections.

We will show that an RTS for an implementation of Ada on highly parallel machines can be written which is free of costly serialization points and contains only short critical sections. The RTS SMARTS depends on the hardware synchronization primitive fetch&ϕ and implements Ada's tasking features in a highly parallel manner. SMARTS is augmented with compile-time support for some implementation-defined pragmas that we propose, and a library of tasking idioms. These idioms, which are variations of the *beacon tasks* described in [Schonberg and Schonberg 1985], will be translated into efficient operations on shared variables on machines with hardware support for fetch&ϕ's.

Furthermore, we have found that the Ada tasking model fits a shared-memory or hybrid multiprocessor quite well, even when the number of processors is large. In SMARTS, Ada shared variables are efficiently implemented (despite their anomalies) using *relay sets*. Relay sets not only provide a means for identifying and resolving references to shared variables, but also facilitate rendezvous optimizations.

We further reduce the overhead of Ada tasking by *micro-tasking*: the explicit scheduling of a family of Ada tasks on a specified number of processors. Thus, Ada tasks are implemented as *lightweight* processes managed by SMARTS, rather than as full-blown operating systems processes.

1.5.1 Structure of the Rest of This Thesis

We discuss the target machine and the operating system support assumed in the design of SMARTS in Chapter 2. In Chapter 3, the synchronization mechanisms and queue algorithms used in SMARTS are given. Chapter 4 represents the bulk of this work – it describes the

SMARTS system itself. The machine dependent optimizations supported by SMARTS are the subject of Chapter 5. In Chapter 6, we assess our implementation of Ada tasking. In particular, we consider how effective SMARTS is at allowing Ada programs to take advantage of their execution on highly parallel machines. The Appendices contain the actual code for SMARTS and the assertions used to prove the absence of race conditions in the code.

1.5.2 Genealogy of SMARTS

In the preceding sections, we placed SMARTS in its proper technical perspective. In this section, we place SMARTS in its proper historical perspective. SMARTS can be thought of as the offspring of the union of two research projects at New York University: The Ultracomputer Project and The NYUAda Project. Since SMARTS builds directly upon this work, we briefly describe these two projects here.

The Ultracomputer Project

The term *ultracomputer* was coined by Schwartz [Schwartz 1980] to describe a technologically feasible approximation of a *paracomputer*. A paracomputer is an idealized model of computation with a very large number of processors connected to a global memory where every processor can access the global memory in a single processor cycle. While theoretically useful, given current technology, a paracomputer cannot be realized. The goal of the Ultracomputer group was to build as close an approximation to a paracomputer as possible using state of the art hardware, as well as to develope the software needed to make effective use of such a machine. The IBM RP3 project was initiated in cooperation with the NYU Ultracomputer project. The architecture of the RP3 is similar to that of the Ultracomputer, although it has several enhancements. The two machines will run a common operating system, SYMUNIX ([Edler *et al.* 1988a], [Edler *et al.* 1988b]), which is being developed by the Ultracomputer group.

The Ultracomputer and RP3 are MIMD machines consisting of a large number of processing elements connected by an Omega network to a large number of memory modules (for simplicity, we will assume that there the same number of processing elements as there are memory modules). The hardware synchronization primitive of these machines is the fetch&ϕ instruction, where ϕ is a binary associative operator, for example, add. The power of the fetch&ϕ is that by combining requests for the same memory location in the switches of the Omega network, the same amount of time is used to perform the operation for a single processor as for many [Gottlieb 1987] (see §2.1.3 for details on combining). However, the number of switches in a path from a processing element to a memory module is logarithmic in the number of processing elements. So while neither the Ultracomputer nor RP3 achieves the paracomputer model of computation, their global memories can be simultaneously accessed by all the processors in log of the number of processors cycles.

Numerous synchronization mechanisms and highly parallel algorithms have been written using fetch&ϕ's. Indeed, fetch&add's (fetch&add is an instance of fetch&ϕ) have been used to develope an entire highly parallel operating system for the Ultracomputer and RP3, SYMUNIX. SYMUNIX is based on the self-service paradigm which relies on the simultaneous distributed processing of centralized data, such as queues of ready-to-run processes, provided by fetch&add's.

Many of the same problems that must be addressed by the designer of operating system targeted to a parallel architecture (for example, scheduling), must also be addressed by the designer of an RTS targeted to a parallel architecture. The highly parallel solutions to these problems developed by the designers of SYMUNIX greatly reduced the task of designing SMARTS.

The NYUAda Project

SETL [Schwartz *et al.*1986] is a very high-level language based on set theory. SETL programs can be written at many different semantic levels. Thus, SETL facilitates software prototyping by allowing a program to be refined into successively lower levels of detail while staying within the language. The NYUAda Project began as a large-scale software prototyping experiment using the SETL programming language. The case study of this experiment was an Ada compiler.

A prototype Ada compiler was rapidly produced using the very high-level features of SETL. The compiler, Ada/Ed, was the first Ada compiler ever to be validated (1983) and became a de facto operational definition of the Ada programming language.

The next phase of the experiment was to improve the performance of Ada/Ed and to bring it closer to an implementation in a more traditional programming language [Kruchten and Schonberg 1984]. This involved rewriting the compiler using the lower-level features of SETL. The tasking module (i.e., RTS) for this low-level version of Ada/Ed was written by Rosen [Rosen 1985].

Once Ada/Ed was written in low-level SETL constructs, it was straightforward to translate Ada/Ed into C. The result of this translation was called Ada/Ed-C. The Ada/Ed-C compiler was developed on a VAX and has been ported to various other uniprocessors (e.g., SUN's and IBM-PC's).

The Ada/Ed-C compiler is being retargetted to two highly parallel shared-memory machines, the NYU Ultracomputer (running the SYMUNIX operating system) and the IBM RP3 (running the MACH operating system [Acetta *et al.* 1986], [Baron *et al.* 1986]). Although the tasking module of the uniprocessor version of Ada/Ed-C supports multi-tasking, it relies on sequences of code being executed serially and without interruption to ensure correctness.

Moreover, no attempt was made to avoid processing tasks serially when implementing RTS functions. Such critical sections and serialization points will degrade performance and result in bottlenecks on highly parallel machines. Thus, porting the Ada/Ed-C compiler to these radically different architectures necessitated a rewrite of portions of the tasking module.

This thesis describes the design of this new tasking module, SMARTS. Since the Ada/Ed-C tasking module and its interface had been throughly tested in a uniprocessor environment, we were able to concentrate on the performance issues inherent to highly parallel machines (e.g., the removal of serial bottlenecks) in the design of SMARTS. SMARTS has been incorporated into the uniprocessor Ada/Ed-C compiler by Dinning [Dinning 1987], and its port to the Ultracomputer prototype is underway.

2. Environment

Although the design of SMARTS was strongly influenced by the architectures of the Ultracomputer and RP3 and their operating system SYMUNIX, an effort was made to keep SMARTS portable. To facilitate the porting of SMARTS to other (similar) machines, we have assumed only minimal operating system support and targeted SMARTS to a generic machine (which is representative of the Ultracomputer and RP3). (Since fetch&ϕ's are integral to the efficiency of SMARTS, SMARTS will, obviously, run better on machines with hardware support for fetch&ϕ's.) Before describing this operating system support and machine model, we give some general thoughts on the types of operating system and architectural support that enable an efficient RTS for parallel programming languages on highly parallel machines to be realized.

2.1 Architecture and Operating System Support for Parallel Languages

The division of labor between the hardware and software and between an operating system and the RTS for a language will determine, in part, how efficiently programs written in the language will execute. For machines with a large number of processors, the trend has been to increase the role of software, since hardware solutions often do not scale up. Software implementations are more flexible than hardware implementations, and thus, better able to deal with the volatile nature of parallel programs running on highly parallel machines.

Efficiency considerations similarly dictate that RTS's for parallel languages on this class of machines usurp some operating system functions. Having an RTS perform operating system functions also reduces its reliance on the underlying operating system. (However, being built atop an inefficient operating system or architecture may adversely affect the performance of an RTS.) In this section, we consider what types of architectural and operating system support are most amenable to an efficient implementation of RTS's for parallel languages on highly parallel machines. Specifically, we consider: operating systems support for parallel languages, operating system/RTS interface, hardware support for operating systems and parallel languages, and hardware/software interface.

2.1.1 Operating System Support

Operating systems on uniprocessors are usually based on the *server/client* model. The operating system is the server and user processes (e.g., the RTS of a compiler) are the clients. When a user process (say the RTS of a compiler) needs a service (say a piece of storage) it calls the corresponding operating system routine to have the service performed. It is often convenient to view the server routines as parallel processes with independent threads of control, e.g., a memory manager server process, a scheduling server process etc., as their execution may be interleaved. On multiprocessors, each operating system server process may actually be executed in parallel. On distributed machines, operating system services are typically requested using messages or RPC's (e.g., [Tanenbaum and Renesse 1985]), whereas on shared-memory machines, server processes can be executed locally (local procedure calls) but must rely on some sort of synchronization mechanism based on shared variables (for example, semaphores) to ensure mutually exclusive access to shared state information (e.g., [Test 1986]).

When the number of processors, and hence, user processes is large, the server/client model breaks down – requesting services of the operating system servers becomes a serial bottleneck. Having more than one server per service may alleviate the situation, however, this measure taken alone is not completely satisfactory since we must still have some means of efficiently and evenly distributing client requests to servers. A viable solution for machines where processors are connected in a hierarchical fashion is to allocate a server per level in the structural hierarchy.

The *self-service paradigm* [Gottlieb 1987] is an alternative to the server/client model. It allows clients to "help themselves" to services. In essence, the RTS functions are *distributed* to the clients, and hence, done in parallel. Clients must coordinate their accesses to shared resources using some sort of synchronization mechanism (which is usually based on shared variables). Efficiently coordinating these accesses is critical; if the clients must serialize their access to shared resources (i.e., they must lock out other clients) then we have bought nothing by moving away from the server/client model – we have traded a server bottleneck for a synchronization mechanism bottleneck. (Hardware support and software techniques for the self-service paradigm are discussed in §2.1.3.)

2.1.2 Operating System /RTS Interface

It may be cheaper for an RTS to replicate certain high-level operating system functions rather than to incur the overhead of system calls. Accordingly, it may be desirable for an operating system to provide low-level primitives that can be used as building blocks rather than more sophisticated functions. After considering several implementations of the high-level parallel programming language LYNX, Scott concluded that the efficiency and flexibility of lower level operating system primitives allowed for a simpler and faster implementation than did higher level operating system primitives [Scott 1986]. He deduced that the ideal operating system from the point of view a language implementor lies at one of two extremes: it either provides everything the implementor needs, or it provides almost nothing, but in a flexible and efficient form. (Anything in between is likely to be awkward and slow he argues.) Since the many idiosyncrasies of Ada make it unlikely that the former – an operating system encompassing all of Ada's tasking features – will be the case, (and if it were our job would be done!) we concentrate on the latter.

The low-level communication and synchronization mechanism provided by an operating system will strongly influence the design of an RTS. To date, the synchronization and communication mechanisms of most operating systems have reflected the underlying hardware. Operating systems for distributed machines usually support message passing and operating systems for shared-memory machines usually support shared variables. RTS's based on message passing often use messages to request that RTS functions be performed remotely (e.g., the Ada RTS's described in [Fisher and Weatherly 1986] and [Jha and Kafura 1985]); RTS's based on shared variables usually use locks (e.g., semaphores) to avoid race conditions while performing RTS functions locally (e.g., the Ada RTS described in [Newton 1987]).

The communication and synchronization mechanisms of a parallel language will have to be implemented by its RTS if they do not match the communication and synchronization mechanisms provided by the operating system. Much of the motivation for implementing global address spaces on distributed machines (e.g., [Bisiani and Forin 1987a] and [Carriero 1987]) was to support parallel programming languages with shared variables On distributed machines, tasks must ultimately send messages to access shared variables, hence, they must know where the variable is currently stored. (Shared variables may migrate with the task that owns them.) The visibility rules of block structured languages such as Ada, may actually facilitate the implementations of shared variables on distributed machines by imposing a hierarchical structure on the owners (tasks) of shared variables (see [Rosenblum 1987])

Scheduling is an example of RTS's replicating what has traditionally been an operating system function for the sake of efficiency A RTS may request non-pre-emptive control over a number of processors from the operating system, and then schedule the parallel tasks of the language on these processors, rather than implementing parallel tasks as full blown operating systems processes ([Cray 1986], [Carnevali *et al* 1986] and [Stone *et al* 1985]) This technique is used in SMARTS and is called micro-tasking There are several advantages to micro tasking First, because the tasks that the processors are to execute have been restricted, the overhead of task creation and allocation is reduced Second, because the processors that will execute these tasks have been restricted, communication costs and contention may be reduced. Finally, micro-tasking avoids the inefficiency that would result from a mismatch of the semantics of tasks and the semantics of operating systems processes. Thus, though counter-intuitive, it can be beneficial for an RTS to duplicate functions already provided by the underlying operating system.

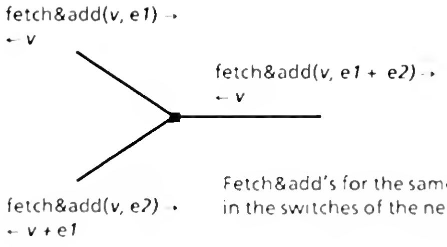
Memory allocation is another operating system function that is a candidate for inclusion in an RTS. It may be more efficient for an RTS to handle its own memory allocation than to use operating system calls. That is, to have the RTS request a large amount of memory from the operating system, and then to allocate this memory to its parallel tasks. The advantages of a RTS self-managing memory are similar to those for micro-tasking.

2.1.3 Architectural Support

Uniprocessor hardware support for the RTS's of sequential programming languages is common (e.g., saving registers upon subprogram invocation). Although the degree of hardware support that is desirable is still not agreed upon, as witnessed by the ongoing RISC (Reduced Instruction Set Computer) vs CISC (Complex Instruction Computer) debate. Direct hardware support on uniprocessors and multiprocessors for the RTS's of parallel languages have also been proposed. Machines have been built with hardware that reduce the cost of task context switches, e.g., register banks where each process has a permanent private set of registers, and remote memory references, e.g., prefetching of data to mask latencies. The shared-memory machine Denelcor HEP began executing a new task (i.e., context switched) on a cache miss, so that processor cycles would not be wasted while the data was being retrieved from the shared memory [Smith 1981]. Again, the utility of some of these proposals has not been demonstrated. An early attempt at an "Ada chip," the Intel 432 [Zeigler *et al.* 1981], turned out to be extremely slow and was a commercial failure. However, as VLSI techniques improve, hardware support for parallel programming languages may become viable. For example, direct support for rendezvous and the queues manipulated by an Ada RTS is being implemented in silicon ([Ardö 1988a] and [Ardö 1988b]). With this special purpose hardware rendezvous are expected to execute in a few microseconds.

The avoidance of serial bottlenecks is a system level problem – it requires cooperation from the RTS, operating system and hardware. For example, it will not be worthwhile for the processors of a shared-memory machine to have data caches (and thereby, reduce the latency of memory references), if maintaining cache consistency requires excessive serialization. Architectural support has been proposed so that even heavy use of the hardware synchronization and communication mechanisms of highly parallel machines will not cause bottlenecks. Specifically, the switches of the network can made to combine either instructions operating on the same shared variable or messages being sent to the same processor. Combining can reduce both network traffic and latency.

Both the Ultracomputer and RP3 intend on combining fetch&ϕ's directed to the same memory location in the switches of their Omega networks. Thus, it will take the same amount of time to perform the operation for a single processor as for many [Gottlieb 1986]. This is demonstrated in the example depicted in figure 2.1.1. When the two instructions, fetch&add(*v*, *e1*) and fetch&add(*v*, *e2*), concurrently issued by processors *p1* and *p2* meet at a switch a single fetch&add(*v*, *e1* + *e2*) is forwarded. When the previous value of *v* is returned



Fetch&add's for the same variable are combined in the switches of the network

Figure 2.1.1 Combining fetch&add's

to the switch, processor *p1* is chosen to receive this value and processor *p2* this value plus *e1*

(Note that *p2* could just as easily been chosen to receive the previous value of *v* and *p1* this value plus *p2*.) Thus, the effect is as though the fetch&add's were performed in some (unspecified) serial order. In our example, fetch&add(*v*, *e1*) is performed first and fetch&add(*v*, *e2*) second. The Connection machine supports a combining send instruction; there is an additional operand in messages which specifies how messages directed to the same destination are to be combined [Hillis and Steele 1986].

2.1.4 Hardware/Software Interface

Since software must ultimately rely on hardware to carry out a function, on a uniprocessor a microcoded (i.e., hardware) implementation of a function will usually be faster than a software implementation of the same function. However, the addition of new functionality to hardware is not free: Due to physical limitations, only a finite number of functions can be implemented in hardware. Thus, it is only worthwhile to support in hardware heavily used functions whose utility has been proven. Furthermore, complex special purpose hardware may slow down other simpler hardware functions (for example, when microcoded instructions are required to complete in a single cycle, adding a complex instruction may increase the completion time of all instructions). Moreover, to take advantage of complex hardware, sophisticated compilers are required (and compile-time optimizations are made more difficult). For these reasons, the trend for uniprocessors has lately been toward simpler hardware (i.e., RISC as opposed to CISC) [Wirth 1987]. (Since it is easier to predict the execution times of programs on a RISC machine where each instructions takes a single cycle to complete, it has also been argued that RISC machines are more suitable for real-time programming.)

The lack of flexibility of hardware may make processor hardware support for complex functions even less attractive for highly parallel machines than for uniprocessors, since the amount of hardware necessary to implement the functions may be proportional to the number of processors and the run-time behavior of parallel programs is more unpredictable than that of sequential programs. In the previous subsection we considered various hardware enhancements that have been proposed to reduce the cost of task scheduling, communication and synchronization. Some of these functions may be more economically performed in software on highly parallel machines.

The division of labor between software and hardware is not uniformly agreed upon by the multiprocessing community. For example, the merits of MIMD (multiple instruction multiple data) vs SIMD (single instruction multiple data) have been widely debated. At issue, is not only what instructions processors execute, but whether processors should run asynchronously or synchronously. (Most MIMD machines run asynchronously, whereas most SIMD machines run synchronously.) The proponents of asynchronous MIMD machines argue that the expense of synchronizing the processors in hardware after each instruction outweighs the savings from not having to synchronize the processors in software (when synchronization is actually required). The optimal hardware/software interface will depend partly on the types of applications that will be run on the multiprocessor. For instance, vector processors are multiprocessors with special purpose hardware for operating on arrays (vectors) of data in parallel. Obviously, the addition of vector processing hardware will only be advantageous if enough application programs contain vector-type parallelism.

Ensuring the cache coherency of highly parallel machines is one area where software is assuming a larger role. The cache coherency of multiprocessors with a modest number of processors can be efficiently maintained by the hardware, e.g., with snoopy caches. However, it is predicted that all (currently known) hardware cache coherency schemes will not scale up [Gottlieb 1987]. Thus, some highly parallel machines rely on the software to ensure cache coherency [McAuliffe 1986].

Interconnection network switch chips that do hardware combining are more complex than those that do not support combining. Hence, the addition of combining hardware may well slow down all global memory references, not just those that need to be combined. Thus, techniques for software combining have been proposed. For example, software combining of fetch&add's is used on the Butterfly [Yew *et al.* 1986]. The Connection Machine also does software combining using a technique called parallel-prefixing [Belloch 1986].

There are two basic approaches to software combining both of which take time that is logarithmic in the number of processes, one for combining operations on shared variables and one for combining messages. To software combine a binary operation on a shared variable, a tree with degree k of shared variables is built. A group of k processes is assigned to each of the leaf nodes of this tree. Each process performs the operation on its assigned node using its own data as the other operand. The process which finishes last goes on to perform the operation on the parent node using the value that was calculated at the child node as the other operand. This last step is repeated until the root node is reached, i.e., the operation has been performed using all the data.

Trees are also used to software combine messages. However, in the message combining case the trees consist of processes rather than variables. Each process only needs to communicate with its parent or children to propagate a message to all the processes in logarithmic time. Some of the more sophisticated techniques for software message combining permit processes to be dynamically added to or deleted from the tree (e.g., [LeBlanc and Jain 1987]).

Software combining techniques are best suited to situations where processes communicate in well-defined, regular patterns, and will only be worthwhile when a large (i.e., at least logarithmic) number of concurrent operations are the norm. Further, note that software combining techniques take a logarithmic number of *software* steps each of which may require a transversal of the interconnection network. Hardware combining techniques take only a single *software* step, i.e., one traversal of the interconnection network.

2.2 Operating System Support for SMARTS

One of the conclusions that was drawn in the preceding section was that a RTS built using lower-level operating system primitives is usually simpler, more efficient and more portable than one built using higher-level primitives. Accordingly, the operating system support assumed by SMARTS, other than fetch&ϕ's, consists of little more than hardware interfaces. For instance, SMARTS micro-tasks, i.e., schedules its own lightweight Ada tasks, thereby, reducing both its dependence on the underlying operating system and the cost of creating, scheduling and destroying Ada tasks. Specifically, SMARTS uses machine instructions or operating system calls for: the creation and non-pre-emptive assignment to physical processors of operating system processes, caching functions, timer functions and memory allocation functions.

Although SMARTS uses only these low-level operating primitives, its effectiveness could still be impaired by an inefficient operating system, particularly by an inefficient memory allocator. SYMUNIX, however, provides highly parallel implementations (based on fetch&add's) of memory allocation functions. Since the SYMUNIX parallel memory allocator is implemented with the same synchronization primitive used within SMARTS, the memory allocator could easily be incorporated directly into SMARTS (further reducing the reliance of SMARTS on SYMUNIX). The parallel garbage collector for Ada developed by Operowsky [Operowsky 1988] is based on fetch&add's and fetch&store's, and is also a candidate for inclusion in SMARTS.

2.3 Machine Model for SMARTS

Most highly parallel machines are the subjects of research projects being conducted at universities. It is clear from the discuss in §2 1, that there is little agreement on what the architecture of the ideal highly parallel machine should look like. Because few highly parallel machines exist and to increase the portability of SMARTS, we have chosen to target

SMARTS to a rather generic hybrid multiprocessor. Since hybrid machines have both local and global memories, they subsume loosely and tightly coupled machines. In this section we describe this generic machine (which is roughly based on the Ultracomputer and RP3).

We assume that each processor of our target machine is paired with a memory module (its local memory), and is also connected to all of the other memory modules (the global memory) via some kind of (unspecified) connection network. In addition, each processor has a local cache. The latency of a memory request by a processor to the global memory is assumed to be significantly greater than a memory request by the processor to its local cache or memory (say on the order of *log*(*P*), where *P* is the number of processors). We assume further that there is hardware support for the simple (but unconventional) indivisible store and increment instructions discussed in §2.3.2. In the next subsection we discuss the data attributes and cache functions that control data residency within the memory hierarchy. Lastly, we assume that each processor has a local clock, as well the existence of global clock (which gives the time of day).

2.3.1 Memory Hierarchy

Our target machine can be viewed as having a three tier memory consisting of a global memory, local memories and caches. How a datum is accessed will determine where it is most advantageous for the datum to reside in the memory hierarchy. The global memory is accessible from all the processors; data that is shared by tasks that are executed on different processors must be kept in the global memory. The cache and local memory of a processor are accessible only from that processor; the private data of a task that is executed exclusively on a single processor can be kept in th local memory of that processor. For example, if Ada tasks are not migratory, that is, a task executes on only one processor throughout its lifetime, then the private variables of a task can be kept in the local memory of the processor on which the task executes.

There are three types of data: instructions, shared data and private data. Instructions and private data can be cached. Some shared data may be cached temporarily. Specifically, shared data that is accessed by either concurrent reads or an exclusive write (CREW) between points at which the processors synchronize may be cached between these synchronization points (assuming that the data is flushed from the caches at the synchronization points.) We will call temporarily cacheable shared variables *synchronous* (since such variables must be accessed synchronously) and non-cacheable shared variables *asynchronous*. Because of the low frequency of synchronous variable stores, it has been suggested [McAuliffe 1986] that for highly parallel machines a store-through caching policy is preferable for synchronous data, while a store-in policy is preferable for private cacheable data stored in the global memory (only a multiprocessor with global memory and local caches is considered). When a store-in caching policy is used, data flushed from the cache must actually be evicted; when a store-through policy data is used, data flushed from the cache need only be invalidated.

As argued in §2.1.4, to facilitate the utilization of caches, a hybrid architecture should permit the software to specify where a datum should reside in the memory hierarchy. In particular, the software should be able to declare data as global (accessed by more than one processor) or local (accessed by only one processor), and as cacheable, temporarily cacheable (between synchronization points) or non-cacheable. Furthermore, the software should be able to request that a particular type of data be flushed from the cache (say, temporarily cacheable data at synchronization points).

We assume in our machine model that there are machine instructions for specifying these attributes of data when it is allocated and for selectively flushing the cache. We will use *allocate(data, location, cacheability)* and *flush(data_type)* to denote these machine instructions, where *location* is either *global* or *local*,

cacheability is either *cacheable*, *marked* (temporarily cacheable), or *noncacheable*, and

data__type is either *all* (the entire cache is flushed) or *marked*.

Only data declared as *cacheable* or *marked* can be cached, and hence, flushed from the cache. A store-in caching policy is used for data declared as *cacheable*, while a store-through caching policy is used for data declared as *marked*. Data declared as *global* is stored in the global memory; only data declared as *local* is stored in the local memories.

Both the Ultracomputer and the RP3 provide system calls to specify these data attributes and to selectively flush these types of data from caches. The global memory of the RP3 is implemented using an address translation scheme called *interleaving* that spreads contiguous data evenly across the shared portion of the memory modules (i.e., the global memory). Interleaving is particularly useful for arrays where each element of the array is accessed by only one processor: Requests for separate elements of the array will be directed at different memory modules (and hence, not create a hot-spot at a memory module) [Brantley and McAuliffe 1985]. Furthermore, the elements of the array can be individually cached.

There are six combinations of data attributes: <*local*, *cacheable*>, <*local*, *noncacheable*>, <*local*, *noncacheable*>, <*global*, *cacheable*>, <*global*, *marked*>, and <*global*, *noncacheable*>. The pair <*location*, *cacheability*> is called the *storage mode* of a datum [Lipkis *et al.* 1987]. The virtual address space of our machine model is composed of segments that have storage modes corresponding to the data storage modes. (On the RP3, it is even possible to specify the storage mode of a single page.) The data of a each storage mode must be allocated in a segment of the same storage mode. The memory mapping unit maps the virtual segments to the appropriate physical segments.

2.3.2 Hardware Synchronization Primitives

The only hardware synchronization primitives we assume are a store and an increment instruction: fetch&add and fetch&store. fetch&add(v, e) indivisibly returns the present value of v and increments v by e. fetch&store(v, e) indivisibly returns the present value of v and stores the value of e in v. This somewhat unusual choice of primitives is influenced by the existence on the Ultracomputer and RP3 of a the atomic operation fetch&ɸ (fetch&add and fetch&store are instances of fetch&ɸ). Because the utility of fetch&ɸ's for writing highly parallel software has been demonstrated (e.g., the highly parallel ready queues discussed in the next chapter), hardware support for these primitives is being included in other proposed machines with a large number of processors. The fluent machine [Ranade 1988], for instance, includes a deterministic version of fetch&ɸ called a multiprefix (deterministic in the sense that the order in which concurrent fetch&ɸ's are combined is predictable).

Fetch&add's and fetch&store's can be implemented in software on machines that do not directly support fetch&ɸ's in hardware. On machines with a global memory, they can be implemented with whatever read-modify-write instruction is supported, and on machines without a global memory, they can be implemented via message passing. For example, the shared-memory MIMD machine Monarch [Rettburg 1987] has three indivisible instructions, *read*, *steal*, and *write* (these instructions are combined in its connection network) with the following semantics: *Read*(v) waits until the tag of a variable v is set to *full* and then fetches the value of v; *steal*(v) waits until the tag of a variable v is set to *full*, fetches the value of v and then marks the tag of v as *empty*; *write*(v, e) stores the value of e in v and then marks the tag of v as *full*. It can be shown [Gottlieb 1987] that a *fetch&store*(v, e) is equivalent to the sequence *steal*(v), *write*(v, e), and that a *fetch&add*(v, e) is equivalent to the sequence *steal*(v, e), $v := v + e$; *write*(v, e) given that each write to v is preceded by a *steal*(v).

Recall that when ɸ is an associative binary operator (e.g., add or store) concurrent fetch&ɸ's can be combined in the network that is used to connect the processing elements to the global memory modules, and thus, take the same amount of time as a single fetch&ϕ. When fetch&ϕ's are implemented in software, concurrent fetch&ϕ's can be software combined so that they complete in logarithmic time.

Add&λ's are a generalization of fetch&add's [Harrison 1986]. They allow processors to perform certain complex (e.g., non-associative) operations with a single fetch&add on a shared variable. The shared variable is divided into bit fields that contain the operands and the state information for the λ operation. By using the proper encoding of the variable into bit fields, add&λ's can be combined in the network in the same fashion as fetch&add's. Add&λ's are implemented by inserting a small number of processors at the memory module end of the omega network which perform the λ operations (rather than simple fetches) on the specified data. Executing the λ operations at the memory module decreases the amount of network traffic needed to perform the λ operation. Furthermore, the code for executing the λ operation at the memory module end of the network is often simpler than the code for executing the λ operation at the processor end of the network (since the λ operation is atomic from the perspective of the issuing processor when it is performed at the memory module end of the network).

Although no actual hardware implementations of add&λ's currently exist, they have been included in the design of the next Ultracomputer prototype[5]. Therefore, while we do not assume hardware support for add&λ's in our machine model, we do note how their existence would affect the relative costs of the queue algorithms (which are based on fetch&ϕ's) that we consider for inclusion in SMARTS. These candidate queue algorithms and the

[5] Thus, the Ultracomputer continues to grow closer to its cousin the RP3 which consists of processor/memory pairs connected to processor/memory pairs: The original Ultracomputer consisted of processors connected to memory modules. Local cache memories were added to the processors in its next incarnation Now a small number of processors are being added to the memory modules.

synchronization mechanisms that they employ are the topics of the next chapter. All of the synchronization mechanisms that we present can be implemented as λ operations.

3. Synchronization Mechanisms and Queue Algorithms

The performance of an Ada RTS will be strongly impacted by the efficiency of its queue handling. Queues are integral to the implementation of most of Ada's tasking features: time management (time chains consisting of tasks that that are waiting on delays – see §4.5), rendezvous (entry queues – see §4.6), task activation and termination (lists of the dependent tasks of master tasks – see §4.3 and §4.4) and scheduling (ready queues – see §4.2). After analyzing a test suite of Ada programs, Ardö identified queuing as the single most important factor in determining the efficiency of an implementation of Ada [Ardö 1988a]. He predicted that hardware support for queue operations would speed up Ada tasking features by a factor of two to five.

The performance of operating systems is similarly dependent on efficient queue handling. Accordingly, many efficient highly parallel queue algorithms have been developed using fetch&add's for SYMUNIX and its compilers ([Dimitrovsky 1988], [Gottlieb *et al.* 1983a], [Gottlieb *et al.* 1983b], [Rudolph 1982] and [Wilson 1988]).

In this chapter, we consider various highly parallel queue algorithms based on fetch&add's and fetch&store's and determine the most suitable of these algorithms for implementing each of the queues manipulated by SMARTS. We begin by describing the synchronization mechanisms that these algorithms employ. Then, after reviewing several queue algorithms based on fetch&add's that appear in the above references, we present some new queue algorithms. One of the new queue algorithms is an extension of one of the algorithms that we review. The other algorithms are based on fecth&store's. Finally, the most efficient implementation for SMARTS's queues are chosen from among these algorithms.

Before presenting the fetch&ϕ implementations of synchronization mechanisms and queue algorithms, we make some general remarks concerning the implementations. We begin by briefly discussing two possible enhancements (Λ&add's and packed shared variables) to the fetch&ϕ implementations given here. After which, we comment on the timing analysis of the fetch&ϕ implementations. We then introduce the code sections (i e., the actual implementations) given in this thesis.

Enhancements to fetch&ϕ's: Add&λ's and Packed Shared Variables.

As noted in the previous chapter, all of the synchronization mechanisms that we present can be implemented as the λ operation of an add&λ ([Harrison 1986], [Harrison 1988]). Thus, the performance of queue algorithms that depend on these mechanisms for synchronization could be greatly improved on our hybrid machine by adding the appropriate hardware to the memory module end of its interconnection network. (Queue operations, e.g., enqueues and dequeues, have also been implemented directly as λ operations; variants of the queue algorithms considered here could also conceivably be operated on with add&λ's.) The time to execute an add&λ instruction is equal to the time to execute a fetch&add (i.e., to route the add&λ request through the network) plus the time to perform the λ operation (at the memory module end of the network). In addition, some code may have to be executed before and after the add&λ to calculate and encode its operands (into a single shared variable), and to decode and interpret its results

Even without special purpose hardware, it may be possible to reduce the number of global memory references (e.g., fetch&ϕ's) required to implement a synchronization mechanism by packing more than one shared variable into the bit fields of a single shared variable Such *packed variables* can be fetched, operated on, and stored in a single network traversal Furthermore, fetch&ϕ's on packed variables can be combined in the network in the usual manner (When a shared variable is packed into a bit field, the range of values that the shared variable can assume is determined by the number of bits in the bit field Thus, packing shared variables may not always be desirable For instance, a shared variable is sometimes used to generate labels (indices) to the processes that are performing an operation in parallel. If such a variable is packed, then the number of processes may be unnecessarily limited.) In [Freudenthal and Pezé 1988], the synchronization mechanisms discussed in this chapter are implemented using fetch&add's that operate on packed shared variables.

For didactic reasons, we do not describe add&λ's or packed variable implementations of synchronization mechanisms here – interested readers are referred to the attendant references. We do give the execution times (in terms of global memory instructions – see below) of the best known packed variable implementations, so that the relative efficiency of queue algorithms that employ these mechanisms can be properly assesses. (Since add&λ's have not been implemented in hardware, we cannot give the execution times of the add&λ implementations.) We note that current implementations using add&λ's and packed variables does not alter the relative costs of these mechanisms, i.e., a mechanism that requires a more complex fetch&ϕ implementation than another mechanism will also require a more complex packed variable or add&λ implementation than that mechanism. However, the absolute efficiency of the implementations of the mechanisms will affect the space/time tradeoffs of the queue algorithms.

Timing Analysis

Two general comments about the timing analysis of the implementations of the synchronization mechanisms and queue algorithms must be made. (These remarks apply to the other implementations given in this thesis as well.) The first comment concerns the expected number of processor cycles that instructions operating on different types of data (e.g., shared vs private) take to complete. The second comment concerns the effect of contention on the execution times of various synchronization primitives.

Global vs Local Memory Accesses

First, it is unrealistic to assume that instructions that operate on different types of data take the same number of processor cycles to complete For example, an asynchronous shared variable will have to be fetched from the global memory, whereas a synchronous shared variable or a private variable may sometimes reside in a cache or local memory. Recall from §2.3, that in our hybrid machine model a global memory reference takes substantially longer than a local memory or cache reference (although techniques such as pipelining or pre-fetching can be used to mask some of the global memory access latency). If we assume that instructions that operate on private or synchronous data usually take a single processor cycle, then it is reasonable to assume that instructions that operate on asynchronous data take a number of cycles that is logarithmic in the number of processors (since they must transverse the interconnection network). Thus, the asymptotic limits of the execution times given in this thesis should be taken to be (in the worst case) in units of $log(P)$ (where P is the number of processors in the multiprocessor)[9].

Blocking vs Busy-Waiting Synchronization Mechanisms

Second, it is difficult to estimate the amount of time spent executing some of the synchronization mechanisms – several of the synchronization mechanisms we present *busy-wait*, i.e., they loop until permission has been granted, rather than blocking and being unblocked when the resource is free. Busy waiting often takes the form (see §3.1): Perform an operation, e.g., decrement a variable. If the operation is successful, e.g. the variable is still positive, then return. Otherwise, undo the operation, e.g., increment the variable, and try again. For all of the synchronization mechanism given in this chapter both busy-waiting and blocking versions exist.

Busy waiting for a resource can potentially waste processor cycles. Furthermore, busy waiting can lead to deadlock. In the presence of pre-emptive scheduling, it is possible for a

[9] A more subtle assumption that we make in our timing analysis is that the time to transverse the network has a known bound. Even when memory requests are hardware combined in the network, because of contention for limited resources, such as wires, requests may sometime be delayed for an unknown duration. Thus, the time taken to transverse the network in the worst case may be the product of a term that is logarithmic in the number of processors and a term that is linear in the number of processes.

process that is busy waiting to be interrupted at some inopportune moment (e.g., before an unsuccessful operation is undone) leading to deadlock. In the absence of pre-emptive scheduling, it is possible for a process that is busy waiting to consume the very processor cycles that could lead to freeing the resource (by the execution of another process), again leading to deadlock. However, busy waiting versions of synchronization mechanisms are simpler than blocking versions which must have some provision for queuing (and hence, dequeuing) blocked processes.

With blocking synchronization mechanisms, processes that block awaiting permission for a resource must be queued or otherwise managed so that they can be unblocked when the resource is free. The primary reason we are interested in synchronization mechanisms is to implement highly parallel queue algorithms – blocking synchronization mechanisms beg the question. Since the overhead of the blocking versions is comparable to the algorithms that we are interested in implementing, we will use busy-waiting versions of synchronization mechanisms (with the caveat that busy waiting should be avoided in situations where the resource will not be obtainable shortly).

To prevent deadlocks, we must ensure *a*) that a process that is busy waiting is not interrupted (i.e., its processor pre-empted by another process) at an inopportune moment and *b*) that the condition being busy waited on eventually becomes true. The first of these requirements can be accomplished by disabling pre-emption while a process busy waits. The second of these requirements can be accomplished by not allowing a process to be pre-empted or to block while it holds the synchronization mechanism. (Note that, even when blocking versions of synchronization mechanisms are used, if processes are allowed to block without first releasing a mechanism, then there is a potential for deadlocks.)

We can temporarily disable pre-emption in either in software (e.g., by setting a flag) or in hardware (e.g., by masking interrupts). (Preventing a process from blocking while it holds a synchronization mechanism is a design criterion enforced by SMARTS.) We will call portions of code where pre-emptions must be disabled *processor critical sections*, since the processor itself can be viewed as a shared resource to which a process must have exclusive access. In SMARTS, pre-emption is disabled when a processor critical section is entered and enabled when it is left. (Processor critical sections are only required when pre-emptive scheduling or interrupt entries are used – see §4.2 and §4.6.)

Although we cannot quantify the amount of time a process must spend spinning when synchronizing using a given mechanism (since it is application dependent), we can give a qualitative measure of the effect of contention on the execution times of queue operations: the amount of time before the becomes available for one process is proportional to, that is, the number of instructions in the critical section protected by the synchronization mechanism. In the presence of contention, the longer the critical section, the more time spent busy waiting.

Comments on the Code Sections

All of the implementations presented in this thesis are given in Ada, since its is assumed that the reader is familiar with that language. (The actual code for SMARTS is written in C.) Tasks, entries and subprograms are written in lower case, while variables are written in upper case. Keywords are written in boldface. For brevity, type and object declarations are sometimes omitted when the type of the object is either easily discernable from the code or has been defined previously.

In the code, we make use of five procedures, *read*, *write*, *add*, *fetch_and_add* and *fetch_and_store*, that operate on shared integers and addresses (access objects). The shared integers are actually the designated objects of access integers. We must pass the procedures access integers (rather than the integers themselves) because Ada scalar variables are passed by copy, i.e., within a procedure to which a scalar variable is passed a local copy of the variable is referenced. (The ARM says nothing about how scalar parameters declared in a pragma SHARED should be passed.)

The five procedures are executed as atomic actions and have the obvious semantics. They are included in the Ada package *global_memory* described in Chapter 5. To ensure that the procedures are performed as atomic actions, they are implemented with accept statements. (The actual code generated for each procedure is a fetch, store, fetch&add or fetch&store, respectively.)

The necessity of executing the procedures *add*, *fetch_and_add* and *fetch_and_store* as atomic actions is clear. The reason for providing the *read* and *write* procedures may seem a bit mysterious, since reads of and writes to integers and addresses are already atomic. The need for these two procedures arises from the inability to name composite objects in a pragma SHARED. Recall that CREW access of shared variables not named in a pragma SHARED between synchronization points is required. It is desirable to be able to access the (scalar or access type) components of composite shared objects asynchronously, i.e., to access them as if they were declared in a pragma SHARED. If accesses to shared variables are restricted to the above procedures, then their required CREW access between synchronization points will be satisfied no matter what pattern of calls are made to *read* and *write*. (Indeed, the variables will be read only when they are passed as parameters. The actual updating is performed within the procedure via a rendezvous.)

read has only a single parameter, the address of an integer variable or of another address. It returns the value (integer or address) stored in the address that given. The other procedures all have two parameters. The first parameter of *write* is the address (location) where its second parameter should be written It is the address of either an integer or of another address. The second parameter is either an integer or an address.

The first parameter of *add* and *fetch_and_add* is the address of an integer, i.e., an access integer, and the second parameter is an integer *Fetch_and_add* returns an integer. (*add* does not return a value.) Like *read* and *write*, *fetch_and_store* is overloaded; it either returns and operates on integers or addresses. The *fetch_and_store* which returns an integer, has the address of an integer as its first parameter (the location to be fetched from and stored in) and an integer as its second parameter (the value to be stored). The *fetch_and_store* which returns an address, has an address as both its first and second parameters.

The dereference of an access object constitutes a read of the access object. Thus, for shared access objects that are components of composite objects (and hence, cannot be named in a pragma SHARED) the procedure *read* should be used whenever the access object is dereferenced. A consequence of this requirement is that the actual parameters of procedures should not involve dereferencing such access objects. Instead, the designated object should be assigned to a temporary variable via the procedure *read* and this temporary should be passed as the parameter. For example, the statement,

```
fetch_and_store(Q.FIRST, Q.FIRST.NEXT);
```

is not executed as an indivisible operation. The compiler will generate three memory references,

```
-- The value (which happens to be an address) in Q.FIRST.
TEMP_ADDR := read(Q.FIRST);
-- The value (which happens to be an address) in Q.FIRST.NEXT.
TEMP_VALUE := read(TEMP_ADDR.NEXT);
-- Perform the operation.
fetch_and_store(Q.FIRST, TEMP_VALUE);
```

While writing this series of *reads* is required by Ada (and may expose subtle assumptions on the indivisibility of instructions that lead to race conditions), to simplify the presentation of the code we will allow actual parameters to be dereferenced shared access objects that are not named in a pragma SHARED in this thesis.

It could be argued that the procedures *read* and *write* are similarly verbose, and hence, should be dispensed with. We insist upon their usage because they allow us to readily distinguish between references to shared and private variables. As discussed above, the time it takes to access shared variables is usually substantially longer than the time to access non-shared variables. Restricting the access of shared variables to the procedures *read* and *write*, emphasizes this time differential.

3.1 Synchronization Mechanisms

In this section we present, in the order of their complexity, the fetch&ϕ implementations of several synchronization mechanisms. We begin with two simple mechanisms, an indivisible test and increment instruction and a semaphore. Two higher-level mechanisms, A/B-locks and group locks, that implement generalized critical sections are then described.

Test&inc and Test&dec

The instruction test&inc increments a variable *v* by an amount *delta* if the resulting value of *v* does not exceed a given number *bound*. The instruction test&dec decrements a variable *v* by an amount *delta* if the resulting value of *v* is not less than a given number *bound*. The fetch&add implementation of these two instructions [Gottlieb *et al.* 1983a] is given below:

```
type SHARED_INTEGER is access integer;

OVERFLOW : constant integer := MAX_INT;
UNDERFLOW : constant integer := MIN_INT;

function test_and_inc(V : in out SHARED_INTEGER; DELTA, BOUND : in integer)
return integer is
OLD_V : integer;
begin
    if read(V) + DELTA <= BOUND then
        OLD_V := fetch_and_add(V, DELTA);
        if OLD_V + DELTA <= BOUND then
            return OLD_V;
        else
            add(V, -DELTA);
            return OVERFLOW;
        end if;
    end if;
end test_and_inc;


function test_and_dec(V : in out SHARED_INTEGER; DELTA, BOUND : in integer)
return integer is
OLD_V : integer;
begin
    if read(V) - DELTA >= BOUND then
        OLD_V := fetch_and_add(V, -DELTA);
        if OLD_V - DELTA => BOUND then
            return OLD_V;
        else
            add(V, DELTA);
            return UNDERFLOW
        end if;
    end if;
end test_and_dec;
```

The packed shared variable implementations of both test&inc and test&dec require only a single network transversal (for the first fetch&add) when there is no overflow or underflow condition (respectively).

Semaphores

Semaphores[6] have been implemented with fetch&add's [Gottlieb 1983a]. In the fetch&add implementation of a binary semaphore (A binary semaphore has only two values, 0 or 1, representing the states free or taken respectively.) given below

```
procedure p_spin(S : in out SHARED_INTEGER) is
begin -- Wait for lock S.
    while test_and_dec(S, 1, 0) = UNDERFLOW loop
        null;
    end loop;
end p_spin;

procedure v_spin(S : in out SHARED_INTEGER) is
begin -- Release lock S.
    add(S, 1);
end v_spin;
```

The fetch&store implementation of the semaphore is,

[6] Recall that a semaphore is a non negative integer variable on which the two atomic operations P and V are defined. $P(S)$ waits until S is greater than zero and then executes $S := S - 1$. $V(S)$ executes $S := S + 1$.

```
type SEMAPHORE_TYPE is (FREE, TAKEN);
type SEMAPHORE is access SEMAPHORE_TYPE;
procedure p_spin(S : in out SEMAPHORE) is
begin  -- Wait for lock S.
    while fetch_and_store(S, TAKEN) = TAKEN loop
        null;
    end loop;
end p_spin;

procedure v_spin(S : in out SEMAPHORE) is
begin -- Release lock S.
    write(S, FREE);
end v_spin;
```

Since the process busy waits for the semaphore, this type of semaphore is often called a spin lock. Both the fetch&add and fetch&store implementations of spin locks require a single traversal of the network in the absence of contention, although packed variables must be used to achieve this performance for the fetch&add version. A big advantage of implementing semaphores with add&λ's is that the spinning can be done at the memory module end of the network, and thus obviates network traversals (note, however, that the processor will still be blocked awaiting the result).

Readers-Readers Locks

In §1.1.2 we discussed the readers-writer problem where CREW access to a shared resource must be enforced. A high-level mechanism CCR (Conditional Critical Regions) was described that could be used quite naturally to impose CREW access to shared resources. The readers-writer problem can also be solved using fetch&add's [Gottlieb *et al.* 1983b].

A problem that is simpler than, but related to, readers-writer is readers-readers: Again there are two classes of actors that need to access a shared resource. The actors within a class can access the resource concurrently, but actors from different classes must exclude each other. One of these classes may have priority over the other. Note that if there is only a single actor in the higher priority class, a readers-readers solution will actually enforce

CREW access. In SMARTS, wherever readers-writer access is required, there is only a single writer. Hence, a readers-readers solution will be sufficient.

In [Gottlieb *et al.* 1983b] a fetch&add solution to readers-readers using *A/B-locks* is presented: Let the class with higher priority be called *A*, and the one with lower priority be called *B*. When an actor of type *A* wants to access the resource, it first sets the *A-lock*, and then waits for the *B*'s which are currently accessing the resource to finish. No *B* will begin accessing the resource, while the *A-lock* is set. When an actor of type *B* wants to access the resource, it waits for the *A-lock* to be released, it will then try to get the *B-lock*. Since *A*s have priority over *B*'s, if an *A* and a *B* concurrently try to get the lock, the *A* will succeed The code for A/B-locks is given below.

```
type ABLOCK is record
    ALOCK, BLOCK : SHARED_INTEGER := new integer'(0);
end record;

procedure b_lock(AB : in out ABLOCK)  is -- Obtain a B lock.
begin
    loop -- Loop until we get the B lock.
        while read(AB.ALOCK) > 0 loop -- Wait for A's to finish.
            null;
        end loop;
        add(AB.BLOCK, 1); -- Try to get a B lock.
        if read(AB.ALOCK) > 0 then -- Too late, an A has already started.
            add(AB.B_LOCK, -1);
        else -- We got the lock
            return;
        end if,
    end loop;
end b_lock;

procedure b_unlock(AB · in out ABLOCK)  is -- Release a B lock
begin
    add(AB.BLOCK, -1);
end b_unlock;

procedure a_lock(AB : in out ABLOCK) is -- Obtain an A lock
begin
    add(AB.ALOCK, 1); -- Don't allow any other B's to start
    while read(AB.BLOCK) > 0 loop -  Wait for current B's to finish
        null;
    end loop
end a_unlock;

procedure a_unlock(AB   in out ABLOCK) is    Release an A lock
```

```
begin
    add(AB.ALOCK, -1);
end a_unlock;
```

The packed shared variable implementation of A/B-locks requires only one global memory reference to obtain an A or B-lock in the absence of contention [Freudenthal and Pezé 1988]. In SMARTS A/B-locks are used to control access to entry and ready queues – enqueuers and dequeuers forming the two classes (In both of these applications only a single dequeuer is active at any given time.).

Group Locks

Critical sections protected by group locks [Dimitrovsky 1988] are executed by *groups* of processes. Any process that is waiting on a group lock will begin executing the critical section within a constant amount of time equal, at worst, to twice the amount of time it takes to enter and execute the critical section (assuming that processes cannot be pre-empted while in the critical section). (Furthermore, the group of processes that currently hold a group lock can be synchronized within the critical section.) We give the code for a simplified group lock below (the code for synchronizing processes within the group lock critical section is omitted).

```
type GROUPLOCK is record
    -- Maximum ticket allowed in critical section.
    GATE: SHARED_INTEGER := new integer'(0);
    -- Maximum ticket issued for entry into critical section.
    ARRIVED : SHARED_INTEGER := new integer'(0);
    -- Maximum ticket of exiting processes.
    DONE : SHARED_INTEGER := new integer'(0);
end record;

G_READJUST : constant integer := ...; -- Ticket numbers calculated mod G_READJUST.

procedure g_lock(G : in out GROUPLOCK) is
begin
    ENTRY_TICKET := fetch_and_add(G.ARRIVED, 1); -- Register for next group.
    loop
    -- Loop until our group is let in.
        GATE_VAL := read(G.GATE);
        exit when => GATE_VAL >= ENTRY_TICKET;
        if (ENTRY_TICKET - GATE_VAL) > G_READJUST then
        -- Lock was readjusted by g_unlock.
           ENTRY_TICKET := ENTRY_TICKET - G_READJUST;
        end if;
     end loop;
  end g_lock;

  procedure g_unlock(G : in out GROUP_LOCK) is
  begin
     EXIT_TICKET := fetch_and_add(G.DONE, 1);
     GATE_VAL := read(G.GATE);
     if EXIT_TICKET = GATE_VAL then
     -- We are the last, so set up next group.
        if EXIT_TICKET > G_READJUST then
        -- Ticket numbers are getting too large, so readjust.
           add(G.GATE, -G_READJUST);
           add(G.DONE, -G_READJUST);
           add(G.ARRIVED, -G_READJUST);
        end if;
        LAST_ARRIVED := read(G.ARRIVED) - 1;
        if GATE_VAL = LAST_ARRIVED then
        -- No new processes have arrived.
           write(G.GATE, LAST_ARRIVED + 1);
        else
        -- Let the next group commence.
           write(G.GATE, LAST_ARRIVED);
        end if;
     end if;
  end  g_unlock;
```

Clearly, this fetch&add implementation of a group lock is more complicated than the fetch&add implementations of either semaphores or A/B-locks. However, by packing shared variables a group lock can be obtained and released with only six global memory references [Freudenthal and Pezé 1988].

The queue algorithms that we describe in the next section rely on either semaphores, A/B-locks or group locks for synchronization The overhead of using these synchronization mechanisms could be reduced by performing them at the memory module end of the network, i e., by using 's where X is a semaphore, A/B lock or group lock Since each of the mechanisms can be implemented as a single add&X instruction [Harrison 1986], the queues algorithms which rely on group locks would then be more competitive with queue algorithms that rely on less expensive synchronization mechanisms.

3.2 Queue Algorithms

There are several queues maintained by SMARTS: time chains, entry queues, lists of dependent tasks and ready queues. These queues can be classified by their number of enqueuers and dequeuers: one enqueuer and (the same) dequeuer, many enqueuers and one dequeuer, and many enqueuers and many dequeuers. Time chains have one enqueuer and one dequeuer; entry queues and dependent task lists have many enqueuers and one dequeuer; ready queues have many enqueuers and many dequeuers. (The rationale for choosing the number of enqueuers and dequeuers for these queues is not always obvious, e.g., why there is a time chain per processor, and is given in the next chapter.) In this section we discuss possible fetch&add and fetch&store based implementations of these queues.

Queue Operations

The queuing operations that must be supported are enqueues, dequeues and, in some cases, interior removals. Items may have to be removed from the interior of time chains (when an event other than a time out occurs) and of entry queues (when an event other than a rendezvous occurs). (We use the term *item* to refer to the information that must be queued, e.g., some representation of a task. The actual elements that compose a queue we call *nodes*. A node will consist of an item or a pointer to a item, and possibly, some bookkeeping information such as a pointer to another node in the queue.) Another implicit operation on queues is (re) initialization. Avoiding race conditions when a parallel accessed queue becomes empty (and hence, needs to be re-initialized) can be tricky (see §3.2.2.1).

Multi-item Queues

Multi-item queues allow *m* identical items to be represented by one *multi-item* node with multiplicity *m* ([Gottlieb *et al* 1983b] and [Wilson 1988])[7]. Thus, only one enqueue is needed to place the *m* items represented by the multi-item node on the queue. Multi-item queues are useful for situations where a large number of similar items must be processed, for example, the activation of a large array of tasks. For an implementation of Ada on highly parallel machines this situation – the processing of a large array of tasks – is expected to occur often. Multi-item queue algorithms should, therefore, be considered when determining the best implementation for ready queues. Accordingly, we present and consider both single-item and multi-item versions of queues. Since it is only necessary to make interior removals from time chains and entry queues which do not contain duplicated items, for simplicity, we do not show how to perform interior removals from multi-item queues.

Degree of Parallelism

Other than the time taken to perform queue operations, one of the major criteria for choosing a given algorithm to implement one of SMARTS's queues will be the degree of parallelism that the algorithm supports, since we want to avoid serial bottlenecks. For example, as noted earlier, ready queues will have many enqueuers and many dequeuers, and therefore, should be implemented with a queue algorithm that supports both highly parallel enqueues and dequeues. On the other hand, entry queues have only one dequeuer (the owner of the entry), but may have many enqueuers (the callers of the entry). Thus, a queue algorithm that serializes dequeues may be an acceptable implementation for entry queues as long as this algorithm supports a large number of parallel enqueues and interior removals.

Space/Time Trade-off

Another performance issue that must be considered when assessing queue algorithms is their space requirement. As usual, there is trade-off between time and space – increased efficiency and parallelism can often be obtained at the expense of storage. For instance, the

[7] In [Gottlieb *et al.* 1983b] and [Wilson 1988] these types of queues are called multi queues. We prefer the term *multi-item queue*, however, since multi queue is sometimes used to refer to *multiple queues*.

pre-allocation of nodes can remove the need for dynamic allocation and deallocation of nodes (and the synchronization that these operations may entail). If enough extra nodes are pre-allocated then nodes can simply be discarded after use, rather than having to allocating a new node for each enqueue and deallocating a node after it is dequeued. Further, when nodes are dynamically deallocated care must be taken to ensure that the node is not deallocated prematurely, i.e., before all accesses to the node have completed.

The following scheme [Lipkis 1985] represents a compromise between pre-allocation and dynamic allocation: pre-allocate a pool of nodes for each task or process; if these nodes get used up, then create new ones dynamically. (Note that in SMARTS queue operations are performed by operating system processes on the behalf of Ada tasks, i.e., Ada tasks are micro-tasked.) If it can be determined that it is safe to reuse a node, then the node can be returned to the pool. This scheme avoids the cost of dynamically allocating nodes until the pool is used up, and the race conditions that might arise when nodes are deallocated, at the expense of some additional storage (the pre-allocated nodes that are never used and the non-deallocated nodes). Since we do not expect memory to be a limiting resource on highly parallel machines[10], we will adopt this scheme (with a pool of one node pre-allocated for each task) when queue algorithms call for nodes to be dynamically allocated.

We give an overview of several existing parallel-accessed queue and multi-item queue algorithms in the next subsection. We then present a modified version of one of these algorithm and some new queue algorithms in §3.2.2. In §3.3, we determine the best queue algorithm for implementing each of SMARTS's queues.

[10] If it is then it will be worthwhile to employ the parallel garbage collector described in [Operowsky 1988].

3.2.1 Previous Work

Various highly parallel queue algorithms based on fetch&add have been developed for the Ultracomputer and RP3. In this section we discuss three of these algorithms, two that have been written for SYMUNIX and one that was written for a LISP compiler. The SYMUNIX queues are related, the second one builds upon the first. Multi-item versions of this second queue and the LISP queue are given.

3.2.1.1 Circular Array

Queues with a known maximum length can be implemented with a circular array of nodes (or pointers to nodes) [Rudolph 1982]: Let *size* be the maximum number of nodes that can be in the queue at one time. Dequeues and enqueues are implemented using the counters, *head* and *tail*. A enqueuer executes e = fetch&add(*head*, 1), and a dequeuer executes d = fetch&add(*tail*, 1). e mod *size* is the slot into which the node can be enqueued, and d mod *size* is the slot from which the node can be dequeued. To avoid overflow the process that discovers that a counter is equal to some threshold value (which must be a multiple of *size*) decrements the counter by that value.

This queue supports the maximal number of parallel enqueues and dequeues, i.e., *size*. However, it is extremely wasteful in terms of space (since storage proportional to the maximum number of items must pre-allocated), especially if the queue is usually not very full. Furthermore, the queue is only applicable to situations where the maximum number of nodes that can be in the queue is known a priori.

The queue can be enhanced so that, at the expense of some additional enqueue and dequeue overhead and loss of parallelism, it is not necessary to reserve space for the maximum number of nodes. We describe this enhanced algorithm below.

3.2.1.2 Array of Linked Lists

Queues of variable length can be implemented with a circular array of singly-linked lists [Gottlieb *et al.* 1983b?]. Each linked list of nodes is protected by a semaphore. A dequeuer or an enqueuer executes a fetch&add of a counter (*head* and *tail* respectively) to determine which list it should access. The process then must gain control of the semaphore associated with that list, after which it can either enqueue to or dequeue from the list. Before an interior removal of a node can be made, the semaphore associated with the linked list that contains the node must be obtained. Special care must be taken with interior removals to ensure that FIFO order is maintained; this is accomplished by including a *missing* field in each node [Wilson 1988].

The number of parallel enqueues, dequeue and interior removals supported by a queue implemented with a circular array of linked lists is equal to the size of the array. The space requirement of the algorithm is equal to the size of the array plus the number of nodes in the queue.

3.2.1.3 Multi-item Array of Linked Lists

Various multi-item queues have been implemented using circular arrays of linked lists [Wilson 1988]. We give one of these multi-item queue algorithms here.

The enqueue of a node containing a multi-item to an array of linked lists can be accomplished in the same manner as the enqueue of a node containing a single-item (described above). The only difference is that the node being enqueued must have a field, *mult*, that is set to the multiplicity of the item.

Dequeues, however, are more complicated A pointer *first* and a counter *firstmult* are kept. *First* points to the first node in the queue and *firstmult* is the number of remaining items represented by the this node To reserve one of the items represented by *first*, a dequeuer must decrement *firstmult* and find the resulting value non-negative. If *firstmult* is zero after being decremented then *first* and *firstmult* must also be updated. While *first* and *firstmult* are being updated the other dequeuers will spin waiting for *firstmult* to have a positive value. (Since dequeuers access *firstmult* rather than *first.mult*, after a node is actually dequeued, i.e. *first* is updated, we can be sure that there are no outstanding dequeue references to the node. Hence, the node can be reused.)

The update of *first*, i.e., the actual dequeue of the multi-item node, is accomplished in a similar manner as the dequeue of a single-item node. A counter (*head*) is incremented (mod *size*) to determine the linked list, *l*, whose head node should next become *first*. After gaining control of the semaphore associated with *l*, *first* can be set to this node and *firstmult* to its multiplicity. (Note that the semaphore must be obtained to avoid contention from enqueuers or removers; other dequeuer are blocked until *first* is updated.)

The enqueue parallelism supported by the multi-item version of a circular array of linked list is the same as the single-item version, the size of the circular array. Since the actual dequeues of nodes (i.e., updates of *first* and *firstmult*) must be serialized in the multi-item case, the dequeue parallelism supported will be equal to the average of the multiplicities of the nodes. The amount of storage required is equal to the size of the array plus the number of nodes in the queue.

3.2.1.4 K-ary Tree

Dimitrovsky developed a parallel queue algorithm based on *k*-ary trees in the course of writing a LISP compiler for the Ultracomputer. We give an overview of this algorithm here, see [Dimitrovsky 1988] for details. Items are kept in the leaf nodes of the tree (but not in the internal nodes). In a *k*-ary tree, all non leaf nodes have *k* children. (Thus, a node in a *k*-ary tree contains *k* pointers to its children.) The leaves are numbered from 0 to *N* (where *N* is some power of *k*). Counters, *head* and *tail*, are maintained for dequeues and enqueues (respectively).

To enqueue a node, *tail* is incremented to determine which leaf the node is to become. Let *n* be this number. The enqueuer then follows pointers from the root node of the tree to the *n*th leaf,where it inserts the node. Determining what pointer to follow at each node is efficiently accomplished by viewing *n* as an integer base *k*; for a node at level $l - 1$ in the tree, the child pointer to follow is given by the *l*th rightmost digit of *n* base *k*. (Note that the root is at level *0*.) The enqueuer may have to wait for some of the pointers on the path from the root to the leaf to be set (to a node by another concurrent enqueuer). (To avoid pre-allocating all of the internal nodes of the *k*-tree, these nodes are created dynamically as described below.) Dequeues are implemented in a similar fashion.

When a node is enqueued it may be necessary for the enqueuer to dynamically create some of the internal (dummy) nodes on the path from the root to the leaf node. Specifically, the enqueuer of the leftmost child (i.e., the child with lowest leaf number) of an internal node must create the internal node. (Since an enqueuer may have to insert up to $log_k(N)$ internal nodes, dynamic allocation will be required for this algorithm unless a pool of $log_k(N)$ dummy items are pre-allocated per process[8].) When a leaf node is dequeued, if the leaf node is the last leaf node to be dequeued from the subtree rooted at an internal node then the dequeuer deallocates the internal node.

To prevent queue overflow the tree must periodically be rebalanced. The tree can be rebalanced when all the nodes in the subtree rooted at the rightmost child of the root node have been deleted – the child pointers of the root node are then simply shifted one pointer to

[8] If all of the internal nodes are pre-allocated for the entire tree, then the storage requirement of the algorithm would be $O(N)$. Thus, the algorithm would have similar storage requirements and support similar degrees of parallelism as the circular array queue algorithm. However, the enqueue and dequeue of an item to a circular array is much simpler than the enqueue and dequeue of an item to a k-ary tree. Thus, the circular array algorithm would perform better

the right.

Each queue operation (enqueue or dequeue) must transverse a branch of the tree, and thus, take $O(1 + log_k N)$ number of steps. The maximum number of enqueues or dequeues that can be done in parallel is N (Enqueues and dequeues do not exclude each other.). The space requirement of this algorithm is $O(1 + log_k N +$ number of nodes currently in the queue). Due to its low storage requirements (compared to the amount of parallelism supported), this algorithm is well suited for situations where the number of items (and hence nodes) in a queue varies widely.

3.2.1.5 Multi-item K-ary Tree

Dimitrovsky extends his k-ary tree queue algorithm to allow for multi-item nodes. Again *head* and *tail* counters are maintained Interior nodes are used to hold multi-items with multiplicity equal to the number of leaf nodes in their subtree.

The enqueue of a multi-item with multiplicity m to a k-ary tree of height h proceeds as follows: The *head* counter is incremented by m to determine the starting (and stopping) node of the item in the tree. Let the multiplicity converted to base k be, $d_n\ d_{n-1}\ \ldots\ d_0$ (where d_0 is the low order digit). For each digit d_i (starting with the high order digit), d_i multi-items (each with multiplicity k^i) are placed in nodes at height $h - i$ in the tree (note that we take the root node to be at height 0). Thus, in the worst case, $log_k(m)$ nodes may have to be placed in the tree for an item with multiplicity m. (In the best case, only one node will have to be placed in the tree, i e., when the subtree rooted at the starting node contains m nodes.)

Dequeues need to be slightly modified from the single-item algorithm, so that they stop before reaching a leaf node when they encounter an internal node with multiplicity

The asymptotic time and space behavior of the multi-item *k*-tree algorithm and the single-item *k*-tree algorithm are similar. However, the multi-item algorithm will outperform the single-item algorithm when there are a large number of items with high multiplicities.

3.2.2 Present Work

The queues we have presented thus far are very general. They attempt to support as many concurrent enqueues and dequeues as possible. As noted earlier, many of the queues manipulated by SMARTS have only a single dequeuer (time chains, entry queues and dependent tasks lists). Following up on an idea suggested by Edler [Edler 1985], we have developed a linked list algorithm based on fetch&store's that supports an unbounded number of concurrent enqueues, while serializing dequeues. Thus, the algorithm is ideal for implementing queues where there is only one dequeuer since all operations on the queue (i.e., multiple enqueues and a single dequeue) can be done in parallel. Furthermore, the algorithm has a very low storage overhead (the amount of storage required for an empty queue) and can be easily modified to accommodate concurrent interior removals. When the algorithm is extended to include multi-items, the degree of parallel dequeues supported is be equal to the average of the multiplicity of items. Therefore, the algorithm may also be appropriate for queues with more than one dequeuer where the number of items on the queue varies widely, provided that many of the items are identical.

In this subsection, we present several versions of this fetch&store linked list queue algorithm. We also present a variation of the *k*-ary tree queue algorithm discussed in the previous section. This new *k*-ary tree algorithm removes the need for the dynamic allocation of interior nodes while only slightly increasing the storage requirements of the algorithm. Thus, the algorithm is even better suited than *k-ary* tree algorithm presented in the previous section for situations where the number of items vary widely. Since the algorithms given in this section are new we discuss them in more detail than those in the previous section.

3.2.2.1 Linked Lists

It is easy to construct linked lists using fetch&store's that support an unlimited number of concurrent enqueues - unfortunately, (due to the race condition discussed below) it is necessary to serialize dequeues. Thus, they are ideal for situations where there is a sole dequeuer. The queue algorithm given in this subsection assumes that there is only one dequeuer. Pointers *first* and *last* to the first and last nodes on the list are maintained. These pointers are updated with fetch&store's as follows:

To enqueue a node, *n*, the instruction *prev* = fetch&store(*q.last*, *n*) is executed, indivisibly making *n* the last node on the queue and determining what node *n* is to follow. *n* is then inserted in the list with a fetch&store(*prev.next*, *n*).

Whole queues (linked lists) can be enqueued in the same manner as a single node: A queue with first node *f* and last node *l* can be enqueued by executing the two instructions: *prev* = fetch&store(*q.last*, *l*); fetch&store(*prev.next*, *f*).

To dequeue a node, the instruction *old_first* = fetch&store(*q.first*, *q.first.next*) is executed, indivisibly advancing the pointer *first* and retrieving the first node on the list. (If more than one dequeuer was allowed, then a race condition would be introduced since the instruction fetch&store(*q.first*, *q.first.next*) actually requires two memory references. To wit, the value of *q.first* could change between the fetch of *q.first* and the execution of the fetch&store.)

After the queue becomes empty, the next enqueuer must set *first* as well as *last*. It will be a dequeuer that discovers when a queue is about to become empty (by virtue of its dequeue). This dequeuer can inform the next enqueuer of the empty queue condition by setting *last* to *null*. An enqueuer of a node *n* that finds *prev* (the return value of fetch&store(*q*.*last*, *n*) equal to *null*, must set *first* to *n*. Care must be taken, however, to ensure that no enqueues are missed, i.e., that there are no nodes enqueued after the dequeuer discovers that the queue is empty, but before the dequeuer resets *first*. A/B-lock's can be used to eliminate this potential race condition. Each enqueuer must obtain a B-lock. When the queue is about to be empty the dequeuer sets the A-lock and waits for the enqueuers to finish. After an A-lock is granted, if the queue is still about to be empty (i.e., there were no enqueues during the time it took to obtain the A-lock), then the dequeuer sets *last* to *null*.

The algorithm can be modified to allow for interior removals, at the cost of potentially having to be allocate nodes dynamically. A field *tag* (that takes on the values *dequeued*, *enqueued* and *removed*) is added to each node. The *tag* is set to *enqueued* when the node is enqueued. The dequeuer of a node *n* tries to claim *n* by executing a fetch&store(*n.tag*, *dequeued*). A remover of a node *n* each tries to claim *n* by executing a fetch&store(*n.tag*, *removed*). The one that receives the return value *enqueued* has claimed the node. If the remover succeeds, then the node is left in the queue as a place holder. After the dequeuer has attempted to dequeue a node, then the node can be returned to the node pool of its owner. (Either the node is no longer needed as a place holder or it has been dequeued.) Otherwise, the next time the owner of the node wants to enqueue an item, it must allocate a new node to hold the item.

This technique of leaving a node in the queue as a place holder after it has been effectively removed is in some sense a form of optimistic execution. We decrease the overhead and increase the parallelism of the queue operations by optimistically assuming that the dequeuer will have processed the node before we need to use it again. Only when the dequeuer has not processed the node will a penalty be paid (a new node must be allocated).

Below we give the code for queues implemented with linked lists using fetch&store.

Code for Linked Lists

-- Data Structures.

```
type TAG_TYPE is (ENQUEUED, DEQUEUED, REMOVED);
type ACCESS_TAG is access TAG_TYPE;
type NODE_REC;
type NODE_PTR is access NODE_REC;
type NODE_REC is record
    VALUE : ITEM_PTR := null; -- Whatever the queue is to contain.
    TAG : ACCESS_TAG := new TAG_TYPE'(DEQUEUED);  -- Not currently on a queue.
    NEXT : NODE_PTR := null;
end record;
type QUEUE is record
    FIRST, LAST : NODE_PTR := null;
    AB : ABLOCK;
end record;
```

-- Queue Operations.

```
procedure enqueue(N : in NODE_PTR; Q : in out QUEUE) is
-- Enqueue a node N onto the end of the queue Q.
begin
    b_lock(Q.AB); -- Enqueues must be protected by a B-lock. see §3.2.2.1.
    if read(N.TAG) /= DEQUEUED then -- The node may be acting as a place holder in a
    queue.
        write(N, new NODE_REC);
    end if;
    write(N.TAG, ENQUEUED);
    PREVIOUS := fetch_and_store(Q.LAST, N); -- Make N last on queue.
    if PREVIOUS = null then   -- The queue was empty, update first.
        write(Q.FIRST, N);
    else   -- Insert N into the queue.
        write(PREVIOUS.NEXT, N);
    end if;
    b_unlock(Q.AB);
end enqueue;

function dequeue(Q : in out QUEUE) return NODE_PTR is
-- Dequeue a node from the beginning of the queue Q and return it.
begin
    loop
    -- Loop until either we find the queue is empty or we get a node.
        if read(Q.FIRST) = null then
            return EMPTY;
        end if;
        if read(Q.FIRST.NEXT) = null then
        -- The queue may be about to be empty
            a_lock(Q.AB); -- Avoid the race condition discussed in §3.2.2.1.
            if read(Q.FIRST.NEXT) = null then
            -- The queue is still about to empty.
                write(Q.LAST, null);
            end if;
            write(Q.FIRST, Q.FIRST.NEXT);
```

```
                a_unlock(Q.AB);
            else
            -- The queue is not empty so dequeue.
                write(Q.FIRST, Q.FIRST.NEXT);
            end if;
            if fetch_and_store(N.TAG, DEQUEUED) = ENQUEUED then
            -- We got the node.
                return N;
            end if;
        end loop;
    end dequeue;

    function remove(N : in out NODE_PTR; Q : in out  QUEUE) return boolean is
    -- Attempt to remove a node from the interior of the queue Q.
    begin
        b_lock(Q.AB);
        case fetch_and_store(N.TAG, REMOVED) of
        when ENQUEUED => -- We got the tag, leave node in queue as place holder.
            b_unlock(Q.AB);
            return true;
        when DEQUEUED =>
            write(N.TAG, DEQUEUED); -- We missed it, but the node can be reused.
            b_unlock(Q.AB);
            return false;
        when REMOVED => -- Some other remover got it, and the node is a place holder.
            b_unlock(Q.AB);
            return false;
        end case;
    end remove;
```

The algorithm supports an unlimited number of concurrent enqueues and interior removals. (The one situation where enqueues and removals will be delayed is when the queue is nearly empty. Obviously, when the queue is nearly empty only a small number of removals would be possible.) The storage overhead (i.e., the storage in excess of the nodes on the queue) of this algorithm is extremely low: two integers (for the A/B-lock) and two pointers (*first* and *last*) and a flag for each node.

3.2.2.2 No-Lock Linked Lists

The algorithm given in the previous section can be improved upon. In particular, the A/B-lock can be dispensed with entirely Recall that the A/B-lock was introduced to prevent the loss of nodes that might otherwise be enqueued after the dequeuer discovers that the queue is

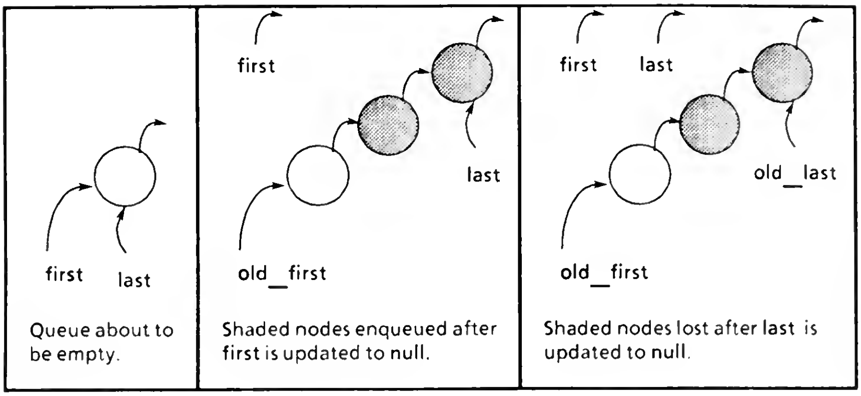

first
last
Queue about to be empty.
first
last
old_first
Shaded nodes enqueued after first is updated to null.
first
last
old_last
old_first
Shaded nodes lost after last is updated to null.

Figure 3.2.1: Potential race condition when a queue is temporarily empty

empty, but before the dequeuer resets *last*. An important observation is that these nodes are not actually lost – only misplaced. Since the old value of *first* was assigned to *old_first*, *old_first.next* will point to the first of the misplaced nodes. If we save the return value of fetch&store(*last*, *null*) in *old_last* (when informing enqueuers of an empty queue condition), then *old_last* will point to the last of these misplaced nodes.

If FIFO order need not be maintained, then the misplaced nodes can simply be enqueued onto the end of the queue. Because the enqueue procedure has two steps (swap the node with *last* and then actually insert the node into the queue), we may have to wait for *old_first.next* to actually be set before enqueuing the misplaced nodes.

If FIFO order is imperative, then we can use an auxiliary queue in addition to the primary queue (i.e., we maintain *aux_first* and *aux_last* as well as *first* and *last* pointers). The misplaced nodes are enqueued onto the auxiliary queue. (Note that enqueuers only access the primary queue.) The dequeuer always attempts to dequeue from the auxiliary queue before the primary queue. Only when the dequeuer finds that the auxiliary queue is empty will it dequeue from the primary queue.

This no-lock linked list can be thought of as a form of optimistic execution. When the list is empty, we update *last* under the assumption that there will be no intervening enqueues. If there were no enqueues, then everything is as it should be. Only when there actually are interfering enqueues must we pay a penalty (i.e., enqueue the nodes that were misplaced).

When the no-lock linked list is used in the array of linked list algorithm discussed in §3.2.1.2 (i.e., we maintain an array of no-lock linked lists) then only dequeues will need to be protected by semaphores. An array of no-lock linked lists will, therefore, support an unbounded number of parallel enqueues and interior removals. However, the number of parallel dequeues supported will still be limited by the size of the array. We can also implement a multi-item queue by modifying the array of no-lock linked list in a manner analogous to what was done for the array of linked list (§3.2.1.2) in §3.2.1.3. As with the algorithm described in §3.2.1.3, the dequeue parallelism will be equal to the average of the multiplicities of the items.

It turns out, that SMARTS has no use for the single-item no-lock linked list. The two SMARTS queues that require single-item linked lists are time chains and entry queues. Time chains have a single enqueuer and dequeuer, and hence, do not suffer from the potential race condition, i.e., they do not need A/B-locks in any case. The entry queues, on the other hand, already need to be protected by A/B-locks for reasons to do with rendezvous management (see §4.5). However, the multi-item version of this improved queue algorithm is an attractive implementation for the dependent lists and ready queues of SMARTS. (Neither of these queues require FIFO order.) Two versions of the multi-item no-lock linked list algorithm are given in the next section.

3.2.2.3 Multi-item Linked List

It is straight forward to alter the no-lock linked list algorithm discussed above to accommodate multi-items – the primary modification is that *first* is only updated when all the items it represents have been dequeued (instead of after each dequeue). Thus, we can relax the restriction that there is a single dequeuer. Like the multi-item queue algorithm presented in §3.2.1.3 (which uses an array of linked lists), with this multi-item queue algorithm the dequeues of the items represented by each multi-item can occur in parallel. However, unlike the multi-item queue algorithm presented in §3.2.1.3, in the algorithm presented here no enqueuers or removers need to be locked out while *first* is being updated. The code for multi-item queues implemented with linked lists is given below.

Code for Multi-item Linked Lists

-- Data Structures.

```
type NODE_REC;
type NODE_PTR is access NODE_REC;
type NODE_REC is record
    VALUE : ITEM_PTR := null; -- What ever the queue is to contain.
    [illegible] : SHARED_INTEGER := new integer'(0);
      [illegible]  NODE_PTR := null;
end record;

type MQUEUE is record
    FIRST, LAST : NODE_PTR := null;
    FIRST_MULT : SHARED_INTEGER := new integer'(0);
end record;
```

-- Queue Operations.

```
procedure enqueue(N : in NODE_PTR; Q : in out MQUEUE) is
-- Enqueue a node N onto the end of the queue Q.
begin
    PREVIOUS := fetch_and_store(Q.LAST, N); -- Make N last node on the queue
    if PREVIOUS = null then    -- The queue was empty, update first.
        write(Q.FIRST, N);
        write(Q.FIRST_MULT, N.MULT); -- Dequeues enabled.
    else    -- Insert N into the queue.
        write(PREVIOUS.NEXT, N);
    end if.
end enqueue;

procedure enqueue_list(FIRST_N, LAST_N : in NODE_PTR;  Q : in out MQUEUE) is
-- Enqueue the list of nodes starting with FIRST_N and ending with LAST_N onto
-- the queue Q.
begin
    PREVIOUS := fetch_and_store(Q.LAST, LAST_N);  -- Make LAST_N last on queue.
    -- Insert the list onto the queue.
    if PREVIOUS = null then   -- The queue was empty, update first.
        write(Q.FIRST, FIRST_N);
        write(Q.FIRST_MULT, FIRST_N.MULT); -- Dequeues enabled.
    else  -- Insert the list into the queue.
        write(PREVIOUS.NEXT, FIRST_N);
    end if;
end enqueue_list;

function dequeue(Q : in out MQUEUE) return NODE_PTR is
-- Dequeue a node from the beginning of the queue Q and return it.
begin
    -- Loop until the queue is empty.
    while read(Q.LAST) /= null loop
        if read(Q.FIRST_MULT) > 0 and then fetch_and_add(Q.FIRST_MULT, -1) > 0 then
        -- We reserved an item in FIRST.
            OLD_FIRST := read(Q.FIRST);
            if fetch_and_add(OLD_FIRST.MULT, -1) = 1 then
            -- We must actually dequeue.
                write(Q.FIRST, Q.FIRST.NEXT); -- Update FIRST.
                if read(Q.FIRST) = null then
                -- The queue is empty.
                    OLD_LAST := fetch_and_store(Q.LAST, null);
                    if OLD_FIRST /= OLD_LAST then
                    -- Some nodes were enqueued, so append them to the end of queue.
                        while read(OLD_FIRST.NEXT) = null loop
                        -- Wait until the first of the nodes is fully enqueued.
                        end loop;
                        enqueue_list(OLD_FIRST.NEXT, OLD_LAST, Q);
                    end if;
                else
                    -- Enable dequeues.
                    write(Q.FIRST_MULT, Q.FIRST.MULT);
                end if
            end if;
            return OLD_FIRST;
        end if;
    end loop;
    -- Queue is empty.
    return EMPTY;
end dequeue;
```

Like the single-item version of the linked list queue, the algorithm supports an unlimited number of concurrent enqueues (and interior removals). However, the actual dequeue of a node still must be serialized. Hence, the dequeue parallelism supported by the algorithm is equal to the average multiplicity of the nodes. Since A/B-locks are not needed, the storage overhead of the algorithm is less than the single item version: one integers (*firstmult*) and two pointers (*first* and *last*) compared to two integers and two pointers for the single item version. (As a consequence of dropping the A/B-locks is that FIFO order is not maintained)

3.2.2.4 Multi-item Linked Lists with Group Locks

In the algorithm presented in the previous subsection, when *first* is being updated, i.e., when an actual node (as opposed to one of the items it represents) is dequeued, other dequeuers are blocked. The dequeue parallelism of the algorithm can be increased by allowing dequeuers to attempt to reserve an item in one of the nodes following *first* as soon as they discover that the multiplicity of *first* has dropped to zero.

If we allow the dequeuers to move on to other nodes, then more than one dequeuer may concurrently reserve the last items of nodes, and hence, attempt to update *first*. In order to reduce the number of nodes at the beginning of the list with non-positive multiplicity, it is the most recent dequeuer that should update *first*. However, determining the precise most recent dequeuer is non-trivial: Because of nondeterministic execution speeds, concurrent dequeuers have to negotiate between themselves to determine the most recent The additional synchronization and overhead required by this scheme becomes quite critical when most nodes have a small multiplicity, since in this case, almost every dequeuer will attempt to update *first*

We can approximate this behavior, i e , determine the "approximate" most recent dequeuer, by using grouplocks Each actual dequeuer (potential updater of *first*) must obtain a grouplock, the node being dequeued by the last dequeuer to join a group is temporarily designated the most recent of this group, after the grouplock has been obtained, if the node following a node being dequeued has a positive multiplicity, then this node is designated as the most recent (note that only one such node can exist per group since the node following it has not actually been dequeued); finally, during the group unlock, *first* is updated with the node following the node that has been designated as the most recent. The code for this multi-item version of the linked list algorithm is given below.

It is not difficult to convince oneself that if all dequeues are executed at the same rate, then this algorithm will outperform the algorithm given in §3.2.2.3 for applications that have a steady stream of concurrent dequeuers and many nodes with large multiplicity. The slight additional overhead when actually updating *first* will be more than compensated for by allowing dequeues to continue during the update. However, since executions speeds may be nondeterministic, it is impossible to determine the average cost of a dequeue with this algorithm: Code executing on different processors will not always execute at the same rate, thus, the last dequeuer to join a group is not always the most recent. For instance, if a process is (temporarily) interrupted at an inopportune moment (i.e., after it has determined that it may have to update *first*, but before it executes the group lock) then *first* could incorrectly be set to an earlier node, this would cause subsequent dequeuers to scan many nodes with negative multiplicity before arriving at the true first node on the queue. Fortunately, such an anomalous event would probably be corrected by the next group of updaters of *first*.

In the absence of pre-emptive scheduling, executions rates of dequeues will be nearly equal (since processors will rarely be interrupted). Thus, this algorithm may be preferable to the fetch&store multi-item queue algorithm discussed in §3.2.2.3 for non-real-time applications (where pre-emptive scheduling is not required).

The data structures and the code for dequeue operation of the multi-item linked list with group locks is given below. The enqueue operation is not given as it is identical to the enqueue operation for the multi-item linked list algorithm given in §3.2.2.3.

Code for Multi-item Linked List with Group Locks

-- Data Structures.

```
    type NODE_REC;
    type NODE_PTR is access NODE_REC;
    type NODE_REC is record
        VALUE : ITEM_PTR := null; -- Whatever the queue is to contain.
        MULT : SHARED_INTEGER := new integer'(0);
        NEXT : NODE_PTR;
    end record;
    type MQUEUE is record
        FIRST, LAST, LAST_IN_GROUP, NEXT_FIRST : NODE_PTR;
        GATE, ARRIVED, DONE : SHARED_INTEGER := new integer'(0); -- For group lock.
    end record;

    G_READJUST : constant integer := ...; -- Ticket numbers calculated mod G_READJUST.
```

-- Group Lock Operations.

```
procedure list_g_lock(N : in NODE_PTR; Q : in out MQUEUE) is
    -- Obtain a group lock prior to dequeuing from queue Q.
    begin
        ENTRY_TICKET := fetch_and_add(G.ARRIVED, 1); -- Register for next group.
        loop
        -- Loop until our group is let in.
            GATE_VAL := read(G.GATE);
            exit when => GATE_VAL >= ENTRY_TICKET;
            if (ENTRY_TICKET - GATE_VAL) > G_READJUST then
            -- Lock was readjusted by g_unlock.
                ENTRY_TICKET := ENTRY_TICKET - G_READJUST;
            end if;
        end loop;
            if ENTRY_TICKET = GATE_VAL then
                write(Q.LAST_IN_GROUP, N);
            end if;
            if read(N.NEXT) /= null and then read(N.NEXT.MULT) > 0 then
            -- Our next node has positive multiplicity, so make it the new first.
            -- (Note: This will only be true for one node per group.)
                write(Q.NEXT_FIRST, N);
            end if;
        end if.
    end list_g lock;

    procedure list_g unlock(Q : in out QUEUE) is
    -- Release a group lock after dequeuing from queue Q.
    begin
        EXIT_TICKET := fetch_and add(G DONE, 1);
        GATE_VAL := read(G.GATE);
        if EXIT_TICKET = GATE_VAL then
        -- We are the last, so update FIRST and set up next group
            update first(Q. LAST IN GROUP).
        if EXIT_TICKET > G_READJUST then
        -- Ticket numbers are getting too large, so re-adjust.
            add(G.GATE, -G_READJUST);
            add(G.DONE, -G_READJUST);
            add(G.ARRIVED, -G_READJUST);
        end if;
        LAST_ARRIVED := read(G.ARRIVED) - 1;
        if GATE_VAL = LAST_ARRIVED then
        -- No new processes have arrived.
            write(G.GATE, LAST_ARRIVED + 1);
        else
        -- Let the next group commence.
            write(Q.LAST_IN_GROUP, null)
            write(G.GATE, LAST_ARRIVED);
        end if;
    end if;
end list_g_unlock

procedure update_first(Q : in out MQUEUE; N : in NODE_PTR) is
-- Actually update the first node pointer of the queue Q.
begin
    -- Update first.
    OLD_FIRST := fetch_and_store(Q.FIRST, N.NEXT);
    if read(Q.FIRST) = null then
    -- The queue is empty.
        OLD_LAST := fetch_and_store(Q.LAST, null);
        if OLD_FIRST /= OLD_LAST then
        -- Some node were enqueued, so append them to the end of queue.
            while read(OLD_FIRST.NEXT) = null loop
            -- Wait until the first of the nodes is fully enqueued.
            end loop;
            enqueue_list(OLD_FIRST.NEXT, OLD_LAST, Q);
        end if;
    end if;
end update_first;
```

-- Dequeue Operation.

```
function dequeue(Q : in out MQUEUE) return NODE_PTR is
begin
    MY_FIRST := read(Q.FIRST);
    while MY_FIRST /= null loop
        M := (fetch_and_add(MY_FIRST.MULT, -1);
        if M < 1 then -- Missed it, try again.
            MY_FIRST := read(MY_FIRST.NEXT);
        else
        -- We have a reserved a node.
            if M = 1 then
            -- We are the last to dequeue a node, may have to update first.
                list_g_lock(N, Q);
                -- Determine if we are latest dequeuer.
                MY_NEXT := read(MY_FIRST.NEXT);
                if MY_NEXT = null or else read(MY_NEXT.MULT) > 0 then
                    -- Note that for each group only one node can have a next node
                    -- with non-zero multiplicity!
                    write(Q.LAST_IN_GROUP, MY_FIRST);
                end if;
                list_g_unlock(Q);
            end if;
        end if;
    end loop;
    return MY_FIRST;
end dequeue;
```

3.2.2.5 K-ary Tree Queue

The algorithms that we presented in §3.2.1 have some undesirable storage properties: Either a large amount of storage must be pre-allocated, and hence, consumed even when the queue is empty (§3.2.1.1, §3.2.1.2 and §3.2.1.3), or nodes must be allocated dynamically (§3.2.1.4 and §3.2.1.5). While, the fetch&store queue algorithms that we described above require very little pre-allocated storage, they only perform well in situations where there is a single dequeuer or there are many identical items (so that multi-items can be used). In this subsection, we discuss a queue algorithm with moderate empty queue storage requirements that does not require the dynamic allocation of nodes and that performs acceptably in a wide variety of situations. Its average case performance is equal to its worst case performance.

The algorithm is a variation of the *k*-ary tee algorithm discussed in in §3.2.1.3 The space requirement of that algorithm is reduced by storing items in the internal nodes of the *k* tree Using actual nodes for internal nodes (rather than dummy nodes) has the added advantage that internal nodes no longer have to be allocated dynamically Furthermore, the bookkeeping necessary when dequeuing items is also reduced While the space requirements, queue operations times and parallelism are asymptotically the same as the original algorithm, the actual amount of storage and time used by the modified algorithm is less

To store items in the interior nodes of a *k*-tree we must impose an ordering on all the nodes of the tree (not just the leaf nodes) Since we can only extract a node from the tree after all of its descendants have been removed we want all the nodes in the subtree rooted at a node numbered n to be numbered lower than n; to minimize the number of dummy nodes (internal nodes inserted before the corresponding item has been enqueued) in the tree we want all the nodes to the left of (and on the same level as) a node numbered n to be numbered less than n (see below). These two objectives can be accomplished by using the following recursive numbering scheme:

For concreteness, consider a k-ary tree of height h containing N nodes. Let c_l be the capacity of a k-ary subtree of height l. c_h is equal to $N - 1$, the capacity of the tree.

$$c_l = \sum_{i=0}^{l} k^i = \frac{(k^{l+1} - 1)}{(k - 1)}$$

1) The root is numbered c_h.

2) The k children nodes of a node numbered n at level $h - l$ are numbered,

$n - 1 - i\,c_l$, $i = 0$ to $k - 1$.

(Note that nodes are numbered from 0 to $N - 1$.) For example, the numbering of a 3-ary tree of height 2 is given in figure 3.2.1.

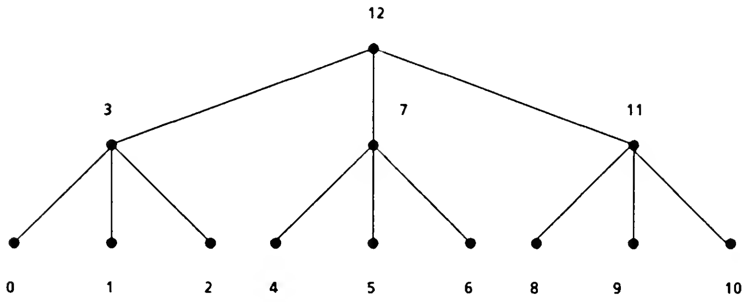


Figure 3.2.1: A Numbered 3-ary tree of height 2.

It is straightforward to incorporate this numbering scheme into the algorithm discussed in §3.2.1.3. Recall that in that algorithm the nth leaf node in the tree is located by following pointers from the root to the leaf node. At each level $l - 1$, the child pointer to follow is the lth rightmost digit of n base k. With our new numbering scheme, at each level $l - 1$, we follow child pointer n *mod* c_l and then divide n by c_l. It is possible to calculate the c_l's on-the-fly, however, the number of calculations per node visited is greatly reduced by storing the c_l's in a table (of size logk(N)).

Having shown how to store and locate items in interior nodes, we are now in a position to modify the algorithm so that dynamic storage allocation is no longer needed. An important observation is that if there are no active enqueues then only $\lfloor log_k N \rfloor$ dummy nodes are needed – one per level in the tree. Thus, by pre-allocating $\lfloor log_k N \rfloor$ nodes we can remove the need for the dynamic allocation of dummy nodes. If we naively apply this scheme, however, then we will limit the potential parallelism since at any given time we may have several enqueuers competing for the dummy nodes. We can eliminate this competition by using a variation of group locks. When we have groups of concurrent enqueuers then it is only the enqueuer to the largest number (position) in the queue that needs to insert dummy nodes along its path – all other nodes along the path will already be in place or will be inserted by members of this group. To take full advantage of the synchronization provided by group locks, during the group lock operation the maximum item being enqueued to the queue (by processes in this group) is calculated and during the group unlock operation a check for possible tree rebalancing is made. Dequeuers do not have to be protected by group locks (since they do not complete for dummy nodes).

Interior removals can also be accommodated by our new k-tree algorithm in a manner analogous to the fetch&store queues: When the owner of a node successfully removes a node, the node is left in the tree as a place holder (see §3.2.2.1 for details).

This modified k-ary tree algorithm supports up to N (the capacity of the tree) parallel enqueuers and dequeuers; however, the enqueues must be executed in groups, i.e., protected by group locks. Furthermore, there is no possibility of starvation since a group lock is guaranteed to be granted within a constant amount of time (equal, at worst, to twice the time taken to execute the code protected by the group lock). Each queue operation, therefore, takes

$O(log_k(N)$ steps. The storage requirements of the algorithm are also $O(log_k(N))$ (the $\lfloor log_k(N) \rfloor$ dummy nodes and subtree capacities).

3.2.2.6 K-ary Tree Multi-item queue Algorithm

We can alter the single-item k-ary queue algorithm given in the previous section to implement a multi-item queue (in similar manner as the single-item k-ary tree queue algorithm presented in §3.2.1.4 was altered in §3.2.1.5 to implement a multi-item queue). As is true in §3.2.1.5, however, the enqueue of a single item with multiplicity may require the enqueue of more than one node. (To facilitate the removal of a multi-item from the interior of K-ary tree, the nodes added to the tree for the item can be linked by *next* pointers.)

The nodes enqueued to the queue (tree) for an item with multiplicity will be of three types (when considered from left to right): leaf nodes that will have their multiplicity set to 1, a series of nodes at higher levels that will that will have their multiplicity set to the number of nodes in their subtree and a final node that will have its multiplicity set to the number of nodes that are free in its subtree. For an item with multiplicity m the minimum number of items enqueued to the tree is 1, the maximum number of items is $O(\lfloor log_k m \rfloor)$ given that $m > k$.

The lower bound is achieved when the node at the position where the item is to be enqueued is the root of a subtree with m free nodes. For the upper bound, consider the worst possible configuration of nodes:

a) k leaf nodes,

b) a series of $k - 1$ nodes at each consecutive lower level with a multiplicity equal to the number of nodes in their subtrees,

c) up to $k - 2$ nodes at the lowest level with a multiplicity equal to the number of nodes in their subtrees, and

d) a final node at the lowest level with a multiplicity of s (whatever is left over).

This is the worst possible configuration since it contains the largest number of smallest subtrees (there can only be k nodes at each level). For example, the worst possible configuration of the nodes enqueued to a 3-tree for an item with multiplicity 9 is shown below.

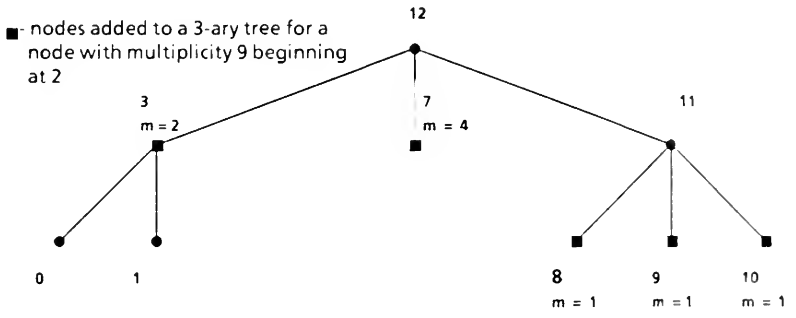

■ - nodes added to a 3-ary tree for a node with multiplicity 9 beginning at 2
12
3
m = 2
7
m = 4
11
0
1
8
m = 1
9
m = 1
10
m = 1

Figure 3.2.2: A Multi-item 3-ary tree of height 2.

The total number of nodes used to represent the item is, $l(k - 1) + 1$, where h is the height of the tree and the first node enqueued for the item is at level $h - l$ (The last node is at level h). The smallest multiplicity of an item that can be represented by this number of nodes using the prescribed configuration will obviously have s equal to one. Thus, this smallest multiplicity m is equal to:

$$\sum_{i=0}^{l} (k-1)c_i - c_l + 2 = m$$

(From a tree with $k - 1$ nodes at each level we add on a 1 to account for case *a*), subtract off a subtree of height l for case c) and add on a 1 for case *d*).) Recall that c_l is the number of nodes in a subtree rooted at level $h - l$ and is given by

$$c_l = \sum_{i=0}^{l} k^i = \frac{(k^{l+1} - 1)}{(k - 1)}$$

Thus, we have,

$$m = \sum_{i=0}^{l} (k^{i+1} - 1) - \frac{(k^{l+1} - 1)}{(k - 1)} + 2$$

Applying the identity a second time gives us,

$$m = \frac{(k^{l+2} - 1)}{(k - 1)} - \frac{(k^{l+1} - 1)}{(k - 1)} - l + 1$$

After regrouping, we are left with

$$m = k^{l+1} - l + 1$$

So, l is $O(log_k(m))$. Now the maximum number of nodes used to represent the item is $l(k - 1) + 1$ (one for each subtree in the worst configuration) which is $O(log_k(m))$ for any constant k.

Code for Multi-item Queue Using a K-ary Tree

-- Data Structures.

```
type NODE_PTR is access NODE_REC;
type NODE_REC(K : in integer) is record -- Tree node
    BRANCHES : array(0..K-1) of NODE_PTR;
    VALUE : ITEM_PTR; -- Whatever the queue is to contain.
    MULT : integer;
end record;

type TREE_QUEUE(K, H : in integer) is  record -- Tree root
    REBALANCE : ABLOCK (others => 0);
    -- For group lock. Note TAIL is equated with ARRIVED.
    DONE, GATE : SHARED_INTEGER := new integer'(0);
    HEAD, TAIL : SHARED_INTEGER := new integer'(1);
    -- Array of capacities of a subtree at each level.
    CAPACITY : array(0..H) of integer; -- Should be constants
    ROOT : NODE_PTR := new NODE_REC(K);
    -- Place holder nodes.
    DUMMY_NODES : array(1..H) of NODE_PTR
                := (others => new NODE_REC(K));
end record;
```

-- Group lock Operations.

```
procedure tree_g_lock(T : in out TREE_QUEUE; MULT : in integer) return integer is
-- Modified group lock for trees.
begin
    N := read(T.CAPACITY(0));
    ENTRY_TICKET :=  test_and_inc(T.TAIL, MULT, 2*N); -- Register for next group.
    if ENTRY_TICKET = OVERFLOW then
    -- Not enough room, try again later...
        return OVERFLOW;
    end if;
    loop
    -- Loop until our group is let in.
```

```
            GATE_VAL := read(G.GATE);
            if GATE_VAL >= ENTRY_TICKET then
                return ENTRY_TICKET;
            end if;
            if (ENTRY_TICKET - GATE_VAL) > N/2 then
            -- Tree and lock were readjusted by g_unlock.
                ENTRY_TICKET := ENTRY_TICKET - N;
            end if;
        end loop;
    end tree_g_lock;

    procedure tree_g_unlock(T : in out TREE_QUEUE; MULT : in integer) is
    begin
        EXIT_TICKET := fetch_and_add(T.DONE, MULT);
        GATE_VAL := read(T.GATE);
        N := read(T.CAPACITY(0));
        if EXIT_TICKET + MULT = GATE_VAL then
        -- We are the last, so set up next group.
            if EXIT_TICKET > N then
            -- Tail and ticket numbers are getting too large, so readjust.
                add(T.GATE, -N);
                add(T.DONE, -N);
                a_lock(T.REBALANCE); -- enqueues are already blocked, block dequeues.
                add(T.TAIL, -N);
                add(T.HEAD, -N);
                write(T.ROOT.BRANCHES(0), T.ROOT.BRANCHES(1));
                write(T.ROOT.BRANCHES(1), null);
                a_unlock(T.REBALANCE);
            end if;
        end if;
        LAST_TAIL := read(T.TAIL) - 1;
        if LAST_TAIL = GATE_VAL then
        -- No new processes have arrived.
            write(T.GATE, LAST_TAIL + 1);
        else
        -- Let the next group commence.
            write(T.GATE, LAST_TAIL);
        end if;
            write(T.GATE, LAST_TAIL);
        end if;
    end tree_g_unlock;
```

Finding Positions in a K-tree

```
    procedure find_largest_enqueue(T : in TREE_QUEUE; POS, MULT   in integer.
                    CURRENT : out NODE_PTR; NEXT, LEVEL : in out integer) is
    -- Find the largest node in a group (and enqueue dummy nodes if necessary)
    begin
        loop
            CAP_LEVEL := read(T.CAPACITY(LEVEL)).
            -- Find the NEXT branch to be taken.
            NEXT := POS div CAP_LEVEL.
            -- Update relative position in subtree
        POS := POS mod CAP_LEVEL;
        -- Loop until CURRENT.BRANCHES(NEXT) is the starting position, or
        -- all the nodes will not fit in the subtree rooted at CURRENT.
        exit when => (POS + MULT >= CAP_LEVEL);
        if read(CURRENT.BRANCHES(NEXT)) = null then
           while (read(T.DUMMY_NODES(LEVEL)) = null) loop
           -- Wait for dummy node to be freed (by a node of this group).
              null;
           end loop;
           write(CURRENT.BRANCHES(NEXT), T.DUMMY_NODES(LEVEL));
           write(T.DUMMY_NODES(LEVEL), null);
        end if; -- Update relative position in subtree.
        -- Update actual node in the tree.
        CURRENT := read(CURRENT.BRANCHES(NEXT));
        LEVEL := LEVEL + 1;
    end loop;
end find_largest_enqueue;

procedure find_enqueue(T : in TREE_QUEUE; POS, MULT : in integer;
                CURRENT : out NODE_PTR; NEXT, LEVEL : in out integer) is
-- Find a position in the tree.
begin
    loop
        CAP_LEVEL := read(T.CAPACITY(LEVEL));
        -- Find the next branch to be taken.
        NEXT := POS div CAP_LEVEL;
        -- Update relative position in subtree.
        POS := POS mod CAP_LEVEL;
        -- Loop until CURRENT.BRANCHES(NEXT) is the starting position, or
        -- all the nodes will not fit in the subtree rooted at CURRENT.
        exit when => (POS + MULT >= CAP_LEVEL);
        while read(CURRENT.BRANCHES(NEXT)) = null loop
           null;
        end;-- Update actual node in the tree.
        CURRENT := read(CURRENT.BRANCHES(NEXT));
        LEVEL := LEVEL + 1;
    end loop;
end find_enqueue;

procedure find_dequeue(T : in TREE_QUEUE; POS : in integer;
                            CURRENT : out NODE_PTR) is
-- Find a position in the tree to be dequeued.
begin
    LEVEL := 0;                  -- Current level in the tree.
    CURRENT := T.ROOT;           -- The address of the root of the tree.
    loop
        CAP_LEVEL := read(T.CAPACITY(LEVEL));
        -- Find the next branch to be taken.
        NEXT := POS  div CAP_LEVEL;
        -- Update relative position.
        POS := POS mod CAP_LEVEL;
        -- Wait for next node to be enqueued.
        while read(CURRENT.BRANCHES(NEXT)) = null loop
           null;
        end;
        -- Update actual node in the tree.
        CURRENT := read(CURRENT.BRANCHES(NEXT));
        if POS = 0 or read(CURRENT.MULT) /= 1 then
        -- Stop, we have reached the position or a node with multiplicity.
            -- May, however, be a dummy node, so wait for value to be set.
            while read(CURRENT.VALUE) = null loop
                null;
            end loop;
            return CURRENT;
        end if;
        LEVEL := LEVEL + 1;
    end loop;
end find_dequeue;
```

-- Queue Operations

```
function initialize(T : in out TREE_QUEUE; N, K : in integer) return is
-- Initialize a K-ary tree T with capacity N.
begin
    write(T.CAPACITY(H), 1); --Note this is the only time CAPACITY is written.
    for I in H - 1..0 loop
        write(T.CAPACITY(I), T.CAPACITY(I+1)*K + 1);
    end loop;
end initialize;

procedure enqueue_node(POS, NODE_MULT : in out integer;
                        CURRENT, NODE : in out NODE_PTR;
                        NUM, LEVEL, NEXT : in integer) is
-- Enqueue one node.
begin
    -- Calculate the multiplicity of this node, and
    NEXT_MULT := min(read(T.CAPACITY(LEVEL)) - POS, NODE_MULT);
    -- the multiplicity remaining to be enqueued.
    NODE_MULT := NODE_MULT - NEXT_MULT;
    if read(CURRENT.BRANCHES(NEXT)) = null then -- Insert ourselves
        write(NODE.MULT, NEXT_MULT);
        write(CURRENT.BRANCHES(NEXT), NODE);
    else --There is a dummy node in my position, so swap nodes.
        write(CURRENT.BRANCHES(NEXT).MULT, NEXT_MULT);
        write(CURRENT.BRANCHES(NEXT).VALUE, NODE.VALUE);
        clear(NODE); -- Set all fields to null.
        write(T.DUMMY_NODES(LEVEL), NODE);
        write(NODE, CURRENT.BRANCHES(NEXT));
    end if;
end enqueue_node;


function enqueue(NODE : in out NODE PTR; VALUE : in ITEM, MULT   in integer,
                T : in TREE QUEUE) return boolean is
-- Enqueue an item ITEM with multiplicity MULT to tree T
NODE_MULT, NEXT   integer
NEXT_NODE   NODE PTR;
begin
```

```
    POS := tree_g_lock(T, MULT);
    if POS = OVERFLOW then
        return FALSE;
    end if;
    NODE_MULT := MULT;      -- Number of items to enqueued.
    NEXT_NODE := NODE;      -- Next node to be enqueued to the tree.
    CURRENT := read(T.ROOT);        -- Current actual node in the tree.
    LEVEL := 0;             -- Current level in the tree.
    if POS + MULT >= read(T.GATE) then
    -- We are the maximum node being enqueued (may have to insert dummy nodes).
        find_largest_enqueue(POS, NODE_MULT, CURRENT, LEVEL, NEXT);
        loop
            enqueue_node(POS, NODE_MULT, CURRENT, NEXT_NODE, LEVEL, NEXT);
            if NODE_MULT = 0 then
            -- We have enqueued all the nodes for the item.
                exit;
            end if;
            -- New relative position in the subtree.
            POS := POS + NODE_MULT;
            find_largest_enqueue(POS, NODE_MULT, CURRENT, LEVEL, NEXT);
            -- Create a new tree node.
            NEXT_NODE := new NODE_REC(T.K);
            write(NEXT_NODE.VALUE, VALUE);
        end loop;
    else
    -- Enqueue a node that is not the maximum in a group.
        find_enqueue(T, POS, NODE_MULT, CURRENT, NUM, LEVEL, NEXT);
        loop
            enqueue_node(POS, NODE_MULT, CURRENT, NEXT_NODE, LEVEL, NEXT);
            if NODE_MULT = 0 then
            -- We have enqueued all the nodes for the item
                exit;
            end if;
            -- New relative position in the subtree
            POS := POS + NODE_MULT;
            find_enqueue(POS, NODE_MULT, CURRENT, LEVEL, NEXT);
            -- Create a new tree node.
            NEXT_NODE := new NODE_REC(T.N, T.K, MULT);
            write(NEXT_NODE.VALUE, VALUE);
        end loop;
    end if;
    tree_g_unlock(T, MULT);
    return TRUE;
end enqueue;

function dequeue(T : in TREE_QUEUE) return NODE_PTR is -
-- Dequeue a node from a given tree.
NODE : NODE_PTR;
begin
    b_unlock(T.REBALANCE);
    POS := test_and_inc(T.HEAD. 1, T.DONE);
    if POS = UNDERFLOW then -- Queue underflow.
        b_unlock(T.REBALANCE);
        return null.
        else
            find_dequeue(T, POS, NODE); -- Find node in tree.
            b_unlock(T.REBALANCE);
            return NODE;
        end if;
    end dequeue;
```

3.3 Implementation of SMARTS Queues

In the last two sections we have presented various queue algorithms. In this section, for each of the queues used by SMARTS (time chains, entry queues, dependent tasks lists and ready queues) we determine which of these queue algorithms should be used to implement it.

The queues to be considered fall into one of three basic classes: array based (§3.2.1.1, §3.2.1.2, §3.2.1.3, §3.2 2.2), linked-list (or fetch&store) based (§3 2.2.1, §3.2.2 3, §3.2.2.4), and K-ary tree based (§3.2.1.4, §3.2.1.5, §3.2.2.5, §3.2.2.6). (Note that the algorithm described in §3.2 2.2 consists of an array of fetch&store linked lists.) These queue algorithms support different numbers of parallel queue operations, take different amounts of time to perform queue operations, and have different storage requirements. We summarize the properties of the queue algorithms in tables 1 and 2. Table 1 compares the best representative single-item queue algorithm from each of the classes we identified above. Table 2 compares the best representative multi-item queue from each class.

Since many of the multi-item queues we have considered perform poorly when most items have a small multiplicity, we include a *combined* single-item and multi-item queue in table 2 A combined queue tries to have the best of both worlds - it combines the best single-item queue (an array of linked lists) with the best multi-item queue (linked list based on fetch&stores), the enqueuers of single items place nodes on the single items queue, while the enqueuers of multi-items place nodes on the multi-item queue, dequeuers can access either queue Thus, combined queues will perform well when there are many identical items, and when there are only a few The disadvantage of combined queues is a slightly increased queue

	Array	Linked List	K-ary Tree
Capacity	∞	∞	N
Dequeue Parallelism	p	1	N
Enqueue Parallelism	∞	∞	N
Time for 1 Dequeue	1	1	$\log_k N$
Time for n Concurrent Dequeues	n	n	$\log_k N$
Time for 1 Enqueue	1	1	$\log_k N$
Time for n Concurrent Enqueues	n/p	1	$\log_k N$
Storage for Empty Queue	p	1	$\log_k N$

Table 1: Comparison of Single-Item Queue Algorithms

N, the capacity of the k-ary tree, is a power of k.
p is the number of linked lists.

operation overhead (due to having to determine which queue to operate on) and the additional storage required by two (as opposed to one) queues. (FIFO order is also sacrificed.)

After making some general observations about the queues summarized in the two tables, we will turn to choosing a queue algorithm for each of SMARTS's queues. We begin with time chains, the simplest SMARTS's queue in terms of the number of parallel operations that can occur (a sole enqueuer and dequeuer). We then discuss entry queues and dependent tasks lists which both can have many enqueuers, but only a single dequeuer. Finally, we tackle ready queues which have many enqueuers and many dequeuers.

	Array	Linked List	K-ary Tree	Combination	
				single	multi
Capacity	∞	∞	N	∞	∞
Dequeue Parallelism	m	m	N	p	m
Enqueue Parallelism	∞	∞	N	p	∞
Time for 1 Dequeue	1	1	$\log_k N$	1	1
Time for n Concurrent Dequeues	n/m	n/m	$\log_k N$	n/p	n/m
Time for 1 Enqueue	1	1	$\log_k N$	1	1
Time for n Concurrent Enqueues	1	1	$\log_k N$	n/p	1
Time for Enqueue of Multi-item	1	1	$\log_k N$	1	1
Storage for Enqueue of Multi-item	1	1	$\log_k m$	1	1
Storage for Empty Queue	p	1	$\log_k N$	p	1

Table 2: Comparison of Multi-Item Queue Algorithms

N, the capacity of the k-ary tree, is a large power of k.
p is the number of linked lists.
m is the average multiplicity of a multi-item.

Table 1 Observations

Because of its low time and storage requirements and simplicity, the linked list based algorithm is clearly preferable to the other algorithms (in the table) for situations where there is a single dequeuer. While both the array based and k-ary tree based algorithms support a high level of parallel enqueuers and dequeuers, there is a clear time/space trade off between the algorithms – the array based algorithm's queue operations require an order of magnitude less instructions than the *k*-ary tree based algorithm's queue operations, but the array based algorithm requires more storage for an empty queue than does the *k-ary* tree based algorithm.

Table 2 Observations

As is true of the algorithms in table 1, the linked list based algorithm has the best storage requirements. The enqueue parallelism of the multi-item version of the linked list algorithm is equal to the average of the multiplicities of nodes. Thus, this algorithm may be appropriate for situations where there is more than one dequeuer, provided that most nodes contain items with a large multiplicity. We see the same space/time trade-off between the multi-item array based queue algorithm and the *k*-ary tree based queue algorithm as we did for the single-item versions. However, because it does not serialize actual dequeues, the *k*-ary tree based algorithm will be clearly superior to the linked list and array based algorithms when most nodes contain single-items. The combined queue is also very robust. There is a space/time trade-off between the combined queue algorithm and the *k*-ary tree based queue algorithms.

3.3.1 Implementation of Time Chains

Before assessing the suitability of various queue algorithms for implementing time chains, we need to know a little about how they are used within SMARTS. As mentioned in §2.2, SMARTS micro-tasks. I.e., SMARTS assigns non-pre-emptable operating system processes to processors; these processes self-schedule the Ada tasks. Each of the non-pre-emptable operating system processes maintains a time chain of the Ada tasks that have executed a delay while being run on its processor (see §4.5 for details). A process will be the sole enqueuer and dequeuer of tasks to its time chain.

Queues with a sole enqueuer and dequeuer pose no special problems in a multiprocessor environment, since operations on such queues are performed serially. For the time chains, however, some provision must be made for other processes to make interior removals, since, for example, a process *P1* may need to cancel a delay set for a task *t* by a process *P2* when *P1* discovers that *t* can engage in a rendezvous. The best (in terms of space and time) queue algorithm that supports parallel interior removals is the linked list based algorithm described in §3.2.2.1. Thus, we will use that algorithm to implement time chains

3.3.2 Implementation of Entry Queues

There is an entry queue for each entry declared by a task. Entry queues can have many concurrent enqueuers (the processes running the callers of the entry), but only a single concurrent dequeuer (the process that is currently running the owner of the entry). Furthermore, items may sometimes have to be removed from the interior of entry queues, for example, when an a entry call is canceled (see §4.6). Because items on entry queues will be single-items, the most suitable queue algorithm for implementing entry queues is the linked list algorithm described in §3.2.2.1 which supports an unbounded number of concurrent enqueuers and removers.

3.3.3 Implementation of Dependent Tasks Lists

Like entry queues, dependent tasks lists can have many concurrent enqueuers (the processes running the activators of the dependents – see §4.3), but only a single dequeuer (the process that is running the master of the dependents during its termination – see §4.4) (Unlike entry queues, items do not have to be removed from the interior of dependent tasks lists.)

Ada programs written for massively parallel machines are expected to create, and hence, activate large arrays of tasks Accordingly, dependent tasks lists should be multi item queues (so that whole arrays of identical tasks can be enqueued as a single node)

The multi-item queue algorithm that seems most appropriate for implementing dependent tasks lists is one of the linked list multi-item queue algorithms (§3.2.2.3 or §3.2.2.4), since they support an unbounded number of concurrent enqueuers.

It is advantageous to implement dependent tasks lists and ready queues with the same algorithm – this allows a dependent tasks list multi-item node to be placed directly on a ready queue when the corresponding array of tasks is being created, activated or terminated (enabling these operations to be performed on the tasks in a bottleneck-free fashion). In the next subsection the linked list multi-item queue algorithms are found to be the best implementation for ready queues in most situations.

3.2.3.4 Implementation of Ready Queues

The number of queues of ready tasks maintained by SMARTS is user defined. There will be a ready queue for each priority level specified (see §4.1 and §4.2). The default number of ready queues is four. When a single priority level is specified, a non-pre-emptive scheduling policy is used (all tasks have the same priority).

As mentioned above, to take full advantage of a highly parallel hybrid machine, a typical Ada program will periodically create large arrays of tasks that operate on a large shared data structure, such as an array or matrix. Thus, the number of tasks in the ready queues is expected to vary widely – both between queues of different priorities and over time. To avoid serializing operations (such as, creation, activation and termination) on arrays of tasks, ready queues should be implemented with a multi-item queue algorithm. While the number of ready tasks at any given time may be large, the number of multi-item nodes on the ready queues will probably be small. Hence, we want queue algorithms that do not consume a large amount of storage when a queue is empty or only has a few nodes.

If all the tasks in an array run until completion, they will never be placed on ready queues as single-item nodes. If, on the other hand, the tasks rendezvous or otherwise block, they will be placed on ready queues as single-item nodes (even though they were created as part of a large array). Recall that SMARTS is augmented with a library of tasking rendezvous idioms that will be translated into efficient operations on shared variables on machines with hardware support for fetch&ϕ's. Tasks that restrict their usage of rendezvous to these idioms can run until completion. Thus, we expect the tasks of most applications written for SMARTS to be able to run until completion.

The two multi-item queue algorithms that do not require a lot of storage for an empty, or nearly empty, queue are the linked list and *k*-ary tree based algorithms (§3.2.2.3, §3.2.2.4 and §3.2.2.5). There are parallelism/time/space tradeoffs between the algorithms, and which of these algorithms is the most efficient for implementing the ready queues of SMARTS is application dependent.

In situations where most tasks do not run until completion, the *k*-ary tree algorithm may be preferable (since, unlike the linked list algorithms, it does not serialize the dequeues of single-item nodes). The long-term, heterogeneous tasks of symbolic and real-time applications may well fit into this category. If group locks are implemented as the λ operation of an add&λ then the *k*-ary tree algorithm becomes even more attractive.

In situations where there are many arrays of tasks that do run until completion, the linked lists based algorithms may be preferable due to their faster queue operations' times and low storage overhead. This is the expected paradigm of scientific code and some artificial intelligence applications (such as low-level vision).

For illustrative purposes, we will use the linked list based algorithm described in §3.2.2.3 to the implementation the ready queues in this thesis. We chose this algorithm, in part, because of its simplicity. More importantly, we feel the algorithm is the most robust. (Recall that, the multi-item linked list queue algorithm described in §3.2.2.5, can behave anomalously when tasks can be temporarily interrupted.) The only drawback of this algorithm is its low dequeue parallelism when there are many single-item nodes, i.e., when tasks do not run until completion.

There are several reason why we do not expect this decreased dequeue parallelism to be crippling. The main situations where task blocking is of concern are when task rendezvous, delay or are pre-empted (since SMARTS creates, activates and terminates large arrays of tasks in a highly parallel fashion). Tasks that rendezvous are already fairly coarse grain (i.e., not on the order of a few machine instructions). (Unfortunately, rendezvous are at best on the order of a few instructions.) A slightly increased dequeue overhead when task rendezvous should not be critical. Moreover, due to the binary nature of rendezvous, the beginning and end of rendezvous by different tasks will most likely be staggered, and therefore, not generate a flood of simultaneous dequeue (or enqueue) requests. Further, real-time applications, where tasks may need to be delayed or pre-empted, will use multiple priority levels. As there is a ready queue for each priority level, the dequeue parallelism will be increased by the number of priority levels. Thus, there is a natural tendency for real-time applications to increase the potential dequeue parallelisms.

Another advantage of the linked list algorithm is that it allows entire linked list to be enqueued in the same manner as a single node. In SMARTS, when a task (whose priority level is undefined) activates a collection of single item and multi item nodes in parallel, a linked list consisting of the nodes is enqueued to a ready queue in this manner.

4. SMARTS

SMARTS is a distributed RTS. A separate incarnation of SMARTS runs on each of the processors that an Ada program is allotted when it begins execution. These incarnations are called SMARTS processors, since they can be considered to be the virtual processors on which the Ada tasks run. SMARTS is based on the self-service paradigm where, rather than having SMARTS processors request that RTS functions be performed by a server process, each SMARTS processor "helps itself" to RTS functions by performing them locally. The SMARTS processors rely on fetch&add's and fetch&store's to coordinate their simultaneous access to shared data (namely, the state information of Ada tasks). In this chapter, we describe the implementation of the services to which the SMARTS processors help themselves.

4.1 Overview of the Implementation of SMARTS

Before moving on to the implementation details of these services, we give an overview of SMARTS. We first define the main shared data structure of SMARTS, the Task Control Block (*TCB*). The *TCB* of a task contains all the information necessary to manage the Ada task throughout its lifetime, i.e., its state. The kernel of SMARTS is then discussed. The kernel is SMARTS's interface to its environment (i.e., architecture, operating system, and user). All of the implementation dependent code of SMARTS is contained in the kernel. The initialization and termination of SMARTS are also the responsibility of the kernel. We end this section with an overview of the subsequent sections of the chapter where the actual implementation of SMARTS processors are described.

4.1.1 Task Control Blocks

Everything that SMARTS needs to know about a task is kept in the *TCB* of the task. For example, to coordinate the termination of the dependents of a task, the *TCB* contains the element *no__term*. (Because the *TCB* of a task completely defines the task, we use the terms *task* and *TCB* interchangeably.) Table 4.1.1*a* and 4.1.1*b* list the various elements of a *TCB*.

Element	Use
abnormal	Flag set when the task is aborted.
block__ptr	Pointer to current block frame.
event	The event which caused the task to become unblocked.
exception	The current exception raised in the task.
master_block	The master block of the task.
master__task	The master task of the task.
mult	The multiplicity when on a ready multi-queue.
next	Pointer to the next task waiting on the same ready or entry queue, or the previous task being serviced by the same task.
no_term	The number of active direct dependents.
num_event	The number of pending events.
sibling	Pointer to the next dependent of task's master construct.
status	The current status of the task.
stack__ptr	Stack pointer.
stack__pos__ptr	Pointer to the top of the stack.
switched	Flag set after the state of the task has been saved.
who	The activator of the task or the caller of an entry of the task.

Table 4.1.1a: Task Control Block Elements for Tasks that run until completion

We distinguish between those elements that are required to manage tasks that run-until-completion (table 4.1.1*a*), and those that are required to manage tasks that use pre-emptive scheduling, delays or rendezvous (table 4.1.1*b*). It is expected that the former elements will be adequate for most scientific applications, but must be augmented by the later elements for most real-time applications. (Note that most of the elements in table 4.1.1*b* are used for rendezvous management.) The elements of the *TCB* are explained in more detail in sections 4.2 through 4.7.

Element	Use
call_priority	Array of counters for the number of tasks at each priority level that are on entry queues of the task.
entry	Pointer to the entry queues of the task.
entry_node_ptr	Pointer to the current entry queue node
num_entries	The number of entries.
open	Array containing currently open entries of the task.
priority	The priority of the task.
rdv	Flag set when the task executes an open accept.
save_priority	Area to save the priority of a task while its executing a rendezvous.
serviced	The tasks currently engaged in rendezvous, as callers, with the task.
template_entry	Pointer to the base of the task template entry table.
template_offset	Offset of the task template entry table.
time_node_ptr	Pointer to the current time_node.
what	The entry that is being called when engaged in a rendezvous.

Table 4.1.1b: Task Control Block Elements for

Pre-emptive Scheduling, Delays and Rendezvous Management

4.1.2 Kernel

Although SMARTS is written in C, there is no theoretical or technical reason why it could not be coded in Ada (The reason that SMARTS was coded in C is that Ada/Ed was already written in C.) To wit, we have translated the C code into Ada for inclusion in this thesis[9]

Given Ada's intended application domain (embedded systems) and tailorability, it not surprising that an Ada RTS can be written in Ada. Writing an Ada RTS in Ada has the advantage that the compiler can be used to bootstrap itself, thereby enhancing its portability. As discussed in Chapter 2, some of the functions performed by an Ada RTS may require operating system or architectural assistance. The Ada library package and pragma

[9] The only significant weakness of Ada for writing RTS's for parallel machines that we encountered was the inability to name composite objects in a pragma SHARED, i.e., the inability to access the elements of composite objects asynchronously. As discussed in § 1.2, this limitation can be worked around - see Chapter 5

mechanisms allow programs to interact with their underlying operating system or architecture. To realize an Ada RTS in Ada, Falis suggests encapsulating interfaces to operating system functions in packages [Falis 1982]. By contrast, Baker proposes that an Ada RTS should be entirely coded in compilable Ada (allowing no interfaces to operating system functions and only a few machine code inserts) [Baker 1987].

The machine code inserts and interfaces to these operating system functions are found in the *kernel* of SMARTS. The interface presented by the kernel to the rest of SMARTS is given in table 4.1.3. As can be seen, SMARTS assumes only minimal architectural and operating

Procedure	Used by	Description
spawn	Scheduler	Procedure used to spawn the SMARTS processors.
fetch_and_add	All	Fetch&add.
fetch_and_store	All	Fetch&store.
flush	Kernel Rendezvous Stack	Flush either all data in the cache at block points, or marked data at synchronization points.
push_state	Scheduler	Save the current state of a task (e.g., its program counter) on its stack.
pop_state	Scheduler	Restore the current state of a task from its stack.
start_timer	Timer	Enable a timer interrupt.
stop_timer	Timer	Disable timer interrupts.
read_timer	Timer	Read the time of day clock.
install	Timer Rendezvous	Install an interrupt handler for a device.
mask_interrupts	Rendezvous	Disable Hardware Interrupts.
unmask_interrupts	Rendezvous	Enable Hardware Interrupts.

Table 4.1.3: Architecture and Operating System interface provided by the Kernel.

system support.

Implicit in the design of SMARTS is the existence of a parallel memory allocator and garbage collector, such as the parallel memory manager of SYMUNIX and the garbage collector presented in [Operowsky 1988]. The kernel must also provide interfaces to the memory management routines. (Since both this memory manager and garbage collector are based on fetch&add's and fetch&store's, they could easily be incorporated directly into SMARTS, removing the need for their kernel interface.)

4.1.2.1 SMARTS Processors

Another aspect of SMARTS's environment to which the kernel must provide an interface is the user. When executing an Ada program, a user of SMARTS can specify three parameters: the number of processors to run the program, the number of priority levels for tasks and the time slice to be used for pre-emptive scheduling. The priority levels and time slice have to do with how the Ada tasks are self-scheduled on the processors. This is the topic of the next section (§ 4.2).

The scheduler contains code to initialize the SMARTS processors with Ada tasks and to detect when the SMARTS processors should terminate (when there are no Ada tasks to schedule). It is the kernel that actually creates the SMARTS processors.

To create the SMARTS processors we use two kernel functions *reserve* and *spawn*; *reserve(num_pe)* requests long term control over *num_pe* processors. *num_pe* is set to the actual number of processors reserved. (The number of processors requested can be specified by the user when a program is executed.) *spawn(num_pe, task)* causes *num_pe* of the reserved processors to begin (non-pre-emptable) execution of the operating system task created from the subprogram *task*. Similar functions are supplied by most multiprocessor operating systems, for example, in SYMUNIX (the operating system of the Ultracomputer) these functions correspond to *reserve(num__pe, 1, WAIT)* and the inclusion of *spawn(num__pe,* NONPREEMPTABLE) as the first statement of the subprogram *task*.

After *num__pe* processors have been reserved, *num__pe* SMARTS processors are spawned.

Element	Use
clock__head	The first time__node on the time chain.
next__slice	The time at which the next time slice will expire.
preempt__lock	Flag that is set to 0 when the processor can be pre-empted.
slice	The time slice being used for pre-emptive scheduling.
task__ptr	The task being executed.

Table 4.1.3: Data Structures for SMARTS Processors

The data structures used by the SMARTS processors are given in table 4.1.3. The SMARTS processors will execute the Ada tasks that are created by the user program. Scheduling is decentralized (i.e., the responsibility of every SMARTS processor) in keeping with the self-service paradigm.

To detect when the SMARTS processors should terminate (perhaps because the program has deadlocked), a global counter, *num__runnable*, is kept of all active tasks, that is, tasks that are currently being executed by a SMARTS processor, tasks that are not currently executing but are ready-to-run and tasks that are suspended at delay statements. The inclusion of tasks that are suspended at delay statements may seem puzzling at first, but they must be considered as active since they can become active without the intervention of an active task, i.e., by a timer interrupt. (For details on the termination of SMARTS processor, see §4.2.3.)

4.1.3 Organization of the Rest of this Chapter

Other than its kernel, SMARTS is composed of eight parts which correspond to the main features of the Ada tasking model: task scheduling, task creation and activation, task termination and destruction, time management, rendezvous management, abort management and storage management. (Time management logically belongs in the kernel as it makes heavy use of operating system functions.) We devote a section in this chapter to each of these parts. Although the amount and nature of the material presented in each of the sections varies, they have roughly the same organization: First we review the ARM rules and any relevant AI's for the feature[10]. The data structures used to implement the feature are then presented. We next describe the implementation of these functions and give their actual code. In some of the sections, after giving the code, we give a rather informal proof that there are no race conditions in the implementation of the feature. We assess our implementation of the Ada tasking features in terms of our goals of efficiency, supporting a high degree of parallelism and minimizing critical sections in Chapter 6.

[10] AI's are Ada Issues that have been brought before the Ada Rapporteur Group (ARG) for clarification. For a listing of approved AI's see [Ada Issues]

4.2 Scheduling

The Ada language reference manual requires that a pre-emptive scheduling policy be used. Ada tasks may specify a static priority with the pragma PRIORITY. The priority of a task indicates its urgency. Thus, priorities are meant to influence the allocation of limited resources to concurrently executing tasks.

The ARM 9.8 (4) states that: *"If two tasks with different priorities are both eligible for execution and could sensibly be executed using the same physical processor and the same other processing resources, then it cannot be the case that the task with lower priority is executing while the task with higher priority is not."*

It has been argued that in a multiprocessing environment it may not be *sensible* to pre-empt an already executing task since context switches may be expensive – in our machine model for example, the network bandwidth consumed while context switching tasks is, in a real sense, a processing resource. Further, determining what task to pre-empt may take a non-negligible amount of time. However, the ARG ruled in AI-00032/08 that a pre-emptive scheduling discipline can not be dispensed with solely on the basis of cost.

Although the above quote from the ARM requires that a pre-emptive scheduling discipline be used for Ada priorities, it does allow some leeway. An implementation is allowed to define the range of priorities that are supported, and in particular, is allowed to not support any priority levels at all [AI-00045]. If only a single priority level is supported by an implementation then all scheduling disciplines will (vacuously) satisfy 9.8 (4). Further, even when an implementation supports multiple priority levels, if all of a program's tasks do not specify priorities or specify the same priority then these tasks can be scheduled according to a non-pre-emptive scheduling policy. To meet the needs of different types of applications (e.g., scientific code), an implementation can supply pragmas for other scheduling policies. Due to its simplicity (and efficiency), run-until-blocked is the default scheduling policy. That is, when none of the tasks of a program specify priorities, the tasks are scheduled with a run-until-blocked policy.

In addition to run-until-blocked and pre-emptive scheduling policies, SMARTS supports a prioritized run-until-blocked scheduling policy. Tasks that wish to be scheduled according to a prioritized run-until-blocked scheduling disciplined must declare SMARTS priorities (and not declare Ada priorities). SMARTS priorities are declared using the pragma SMARTS_PRIORITY. With a prioritized run-until-blocked scheduling policy, a SMARTS processor always chooses the task with highest SMARTS priority to execute. This task continues running until it voluntarily gives up its SMARTS processor because it must wait for an event, i.e., is blocked (In contrast, with a pre-emptive scheduling policy, a task continues running until it is pre-empted or must blocked.)

4.2.1 Overview of Scheduling Policy Implementation

To implement prioritized run-until-blocked or pre-emptive scheduling, a parallel-accessed ready queue for each Ada or SMARTS priority level is maintained With SMARTS priorities (or no priorities), a SMARTS processor runs a task until the task becomes blocked, at which time the SMARTS processor obtains the ready-to-run task of highest SMARTS priority from one of these ready queues. Thus, priorities are respected in the sense that when a task becomes blocked the ready-to-run task with the highest priority is chosen to be executed next

Pre-emptive scheduling is implemented in an efficient bottleneck-free manner using time-slicing Each SMARTS processor is interrupted at periodic intervals where it checks the ready queues for a ready-to run task with a higher priority than the task that it is currently running If such a task is found, then the SMARTS processor begins running the new task (i e , the SMARTS processor is pre empted) Otherwise, the current task is resumed When executing an Ada program, a user can specify the time slice is by setting the appropriate parameter. The default time slice is 50 microseconds.

The responsiveness of SMARTS, i.e., the time for a newly executable task of priority *p* to pre-empt the SMARTS processor of an already executing task of lower priority, will be bounded by the sum of four terms:

- the time to execute the longest processor critical section,
- the time slice,
- the time to execute the timer interrupt handler,
- and the time to (attempt to) dequeue a task from $max\_priority - p + 1$ ready queues.

(Assuming an instruction on our target machine takes 100 nanoseconds, back-of-the-envelope calculations suggest that with a 50 microsecond time slice SMARTS should be able to guarantee that the above sum is less than 100 microseconds – see Appendix A.)

In SMARTS, the only processor critical sections are queue operations that either disable interrupts or busy-wait. In particular, the operations on time chains, ready queues, and entry queues are processor critical sections. The entry queue operations contain the code to initiate rendezvous, and hence, are quite a bit longer than the entry queue operations presented in §3.2.2.1. The enqueue operations of the no-lock multi-item queues described in §3.2.2.3 and §3.2.2.4 does not busy-wait; however, it must be implemented as a processor critical section because the dequeue operation may have to busy-wait for its completion (after an empty queue condition).

It does not affect the response time to disable interrupts while dequeuing from a ready queue, since a SMARTS processor is currently attempting a context switch. (Indeed, we would need to disable interrupts during the actual context switch in any case, lest we risk leaving a task in an inconsistent state.) Similarly, it does not affect the response time to disable interrupts while dequeuing from a time chain, since a SMARTS processor will be pre-empted if its time slice is up.

A simple flag *preempt_lock* belonging to the SMARTS processors is used to software disable interrupts while the SMARTS processor enqueues a task to ready queue or attempts to find a new ready-to-run task. We can use software, rather than hardware, disabling of interrupts because ready enqueue operations do not busy-wait: when interrupts are not hardware disabled, it is possible for an interrupt entry call issued by a device to (temporarily) pre-empt the SMARTS processor. While it is executing on the SMARTS processor, the interrupt entry call may need to enqueue the owner of the interrupt entry to a ready queue. Because ready enqueues do not entail busy-wating there is no possibility of deadlock when a SMARTS processor is (temporarily) interrupted while performing a ready queue operation.

Protecting entry queue operations from pre-emption is more difficult than protecting ready queues. To ensure that an interrupt entry call issued by the hardware does not (temporarily) pre-empt a SMARTS processor from a task that was executing an entry queue operation, hardware disabling must be used. Interrupt entry calls issued by the hardware must similarly be executed as processor critical sections. The code for interrupt entry calls is roughly the same length as the code for non-interrupt entry calls, and hence, will not contribute to the response time of SMARTS. Although the accept statement of the interrupt entry, i.e., the interrupt task, may be executed directly by the hardware, we do not need to add its execution time to our response time as the interrupt task is, in essence, pre-empting the SMARTS processor.

To implement the entry queue operation processor critical sections, we use the kernel functions *mask_interrupt* and *unmask_interrupt* which execute machine instructions for masking and unmasking interrupts (respectively). These instructions exist in some form on every machine. (Recall that, processor critical sections, and hence these functions, are only needed when a pre-emptive scheduling policy or interrupt entries are used.) In the code given in this section, we also make use of the kernel functions *flush(data__type)* described in §2.3.1 to the flush the cache when an Ada task context switch is made on a SMARTS processors (i.e., when the current task is switched out and a new task switched in). (Recall that *data__type* is either *all*, the entire cache is flushed, or *marked*, only the marked data is flushed.)

In the next section we give the data structures that are used to schedule tasks in SMARTS. In §4.2.2, we show how the distributed scheduling of tasks is accomplished using the *event* and *status* fields of *TCB's*. The actual code used to implement task scheduling is discussed in §4.2.3 and listed in §4.2.4.

4.2.2 Data Structures for Task Scheduling

The elements of the *TCB* involved specifically with the scheduling of Ada tasks are fields: *event, mult, next, num__event, status* and *switched*. *event, num__event* and *status* are used to coordinate the blocking and unblocking of tasks. The *switched* flag is also involved with blocking tasks; it is set after the context of a blocked task has been saved. *next* and *mult* are used to implement ready queues.

There is a ready queue for each Ada or SMARTS priority specified (the default number of queues is four). The ready queues are multi-item queues that consist of *TCB*'s (single-item nodes) and *SMARTS__nodes* (multi-item nodes). A *SMARTS__node* contains (pointers to) the *TCB*'s of an array of tasks. *SMARTS__nodes* allow us to perform operations on large arrays of tasks in parallel. The ready queues are implemented with the multi-item queue algorithm described in §3.2.2.3 (see §3 3.4 for a discussion of why this implementation was chosen).

Each SMARTS processor has four data structures used specifically for scheduling: *next__slice, preempt__lock, slice*, and *task__ptr*. *slice* is the length of the time slice being used for pre-emptive scheduling, and *next__slice* the time at which the current time slice is to expire. *preempt_lock* is used to software disable interrupts. *task_ptr* points to the *TCB* of the task that the SMARTS processor is currently executing.

Finally, the system wide counter *num_runnable* is used to detect when SMARTS should terminate, i.e., when there are no active Ada tasks.

In the next subsections we describe *SMARTS_nodes* in detail. We then explain how the *event*, *num_event* and *status* fields of *TCB*'s are used to block and unblock tasks in a highly-parallel fashion.

4.2.2.1 SMARTS_nodes

The Ada language specifies when tasks may be activated in parallel. To avoid serial bottlenecks, in SMARTS we perform other functions on arrays of tasks, such as *TCB* initialization, in parallel. When a task creates an array of tasks (or a pointer to an array of tasks) a *SMARTS_node* with multiplicity (equal to the number of tasks in the array) is created. This *SMARTS_node* will be placed on a ready queue, so that the SMARTS processors will initialize the *TCB's* of the array of tasks in parallel. The *TCB_ptrs* field of a *SMARTS_node* is an array of pointers that are set to point to the *TCB*'s of the tasks in the array. A *SMARTS_node* will be placed on the dependents chain of the master construct of the tasks, and will also be placed on a ready queue to activate and terminate the tasks in a similar parallel fashion[11].

When it is necessary for a task *t* to wait for a function to be performed in parallel on an array of *m* tasks, *t* decrements the *num_event* counter of its *TCB* by *m* prior to placing the *SMARTS_node* for the array of tasks on a ready queue. (The *num_event* counter of the *TCB* of a task counts the number of outstanding or pending events of the task - see §4.2.2.4.) After

[11] In Ada, *SMARTS_nodes* and *TCB*'s must be declared to be of same variant record type. Since the statuses of *TCB*'s and *SMARTS_nodes* are distinct, the *status* field would be the discriminant of this variant record.

performing the function on one of the tasks, a SMARTS processor will increment *t.num__event* (*t* has one less outstanding event).

Since a *SMARTS__node* is treated as a task by the scheduler and master construct of the array of tasks, it must contain some of same the fields as a *TCB*, in particular, those that are needed for ready queue and dependent queue operations: a multiplicity count *mult*, a *next* pointer, a *sibling* pointer and a *status*. The *status* of a *SMARTS__node* gives the function (e.g. initiate, activate) that is to be performed on the array of tasks. In addition, a *SMARTS__node* has two fields specifically used to initialize the Ada tasks that will be created from the *SMARTS__node*: a pointer *parent* to the activator of the array of tasks, and a pointer to the task type template of the array of tasks *template__ptr*.

The elements of a *SMARTS__node* are given in table 4.2.1. Because *TCB*'s and

Element	Use
mult	The multiplicity of the item (i.e., the number of tasks in the array).
next	Pointer to the next task waiting on the same ready queue.
parent	Pointer to the parent of the array of tasks
sibling	Pointer to the next dependent of the master block of the array of tasks.
status	The current status of the SMARTS__node, i.e, the function being performed on the array of tasks
tcb__ptrs	Array of pointers to the TCB's of the array of tasks
template__ptr	Pointer to the type template of the array of tasks

Table 4.2.1 - SMARTS__node

SMARTS__nodes may be accessed by more than one task (actually by more than one of the SMARTS processors that are executing the Ada tasks) they are kept in the global memory.

4.2.2.2 Events and Status

The *event* and *status* fields of the *TCB* of a task are closely related. The *status* of a task is its current activity or state. The *event* of a task is the action that causes a blocked task to become unblocked so it can engage in an activity. The *status* of a task is set only by the task itself (except when the task is participating in a a termination wave where it may be set by one of the dependents of the task – see §4.4.8). Other tasks inspect the *status* of a task when they want to engage in an activity with that task. The *event* of a task is set by other tasks or by a time out. A blocked task inspects its *event* when its execution is resumed. The number of outstanding or pending events for a task is recorded in the *num_event* counter of the task. The *num_event* counter is used to determine when a task should block or be unblocked, i.e., a task is unblocked when the last of its outstanding event occurs (see the next subsection).

In table 4.2.2, we list the possible transitions from *status* to *status* a task undergoes either because of the occurrence of an *event* or because of the execution of a statement[12]. These transitions are diagrammed in figure 4.2.1. As can be seen, tasks live a fairly complicated life. However, most of the states in the diagram are equivalent, in the sense that a task is blocked waiting for an event that would allow it to resume execution, i.e., for an event other than *terminate*. For long-lived tasks this is the normal state of affairs. Since scheduling is mainly concerned with the blocking and unblocking of tasks, in figure 4.2.2 we diagram this

[12] Note the transitions given in table 6 are only those that are required by the ARM. A desirable transition that is not required by the ARM (and thus not included in the table), is for a task to block while it awaits the completion of an I/O operation (allowing another task to run on the SMARTS process). The task should be unblocked by the I/O device handler when the I/O operation is complete, i.e., when the device issues an interrupt. In SMARTS, delays are implemented in a similar fashion: a task blocked at a delay is unblocked by the timer handler after a timer interrupt has been generated. Unblocking tasks upon the completion of an I/O operation is even simpler than unblocking a task after a timer interrupt because blocking for I/O is not an abort synchronization point (see § 4.5 and § 4.7). However, we do not provide such an implementation as we do not cover Ada I/O operations in this thesis.

Status	Event	New Status	Comment
activating	activated	active	All the tasks are activated.
		completed (abnormal)	The task is aborted.
active		complete_block*	The task completes execution of a master block.
		completed	The task completes its execution.
		completed (abnormal)	The task is aborted.
		select	The task executes an accept statement.
		select_term	The task executes a select with a terminate alternative.
		called_rdv	The task executes an entry call.
		timed_rdv	The task executes a conditional entry call.
call	abort	completed (abnormal)	The task will have been removed from the entry queue.
	end_rdv	active	The rdv was successful. (If the owner was aborted then the *exception* field of the TCB will be set.)
complete_block*	terminate	complete_block*	All dependents of a master block are quiescent.
	done	active	All dependents of a master block are terminated.
completed	abort	completed (abnormal)	The task has been aborted.
	terminate	terminated	All dependents of the task are quiescent.
completed (abnormal)	done	terminated	All dependents of the task are terminated
creating	created	active	All the tasks are created
select	abort	completed (abnormal)	The rendezvous will have been disabled
	start_rdv	active	The time out will have been disabled.

Table 4 2 2 - Task State Transitions

Status	Event	New Status	Comment
term_select	abort	completed (abnormal)	The rdv will have been disabled
	start_rdv	active	If a task is called, then it is not quiescent.
	terminate	terminated	All the tasks that depend on a completed master are also quiescent.
terminated	done	terminated	All dependents have freed the TCB's of their dependents
timed_select	abort	completed (abnormal)	The rendezvous will have been disabled.
	start_rdv	active	If a task is called, then it is not quiescent.
	time_out	active	The rendezvous will have been disabled.
timed_call	abort	completed (abnormal)	The task will have been removed from the entry queue
	end_rdv	active	The time out will have been canceled.
	time_out	active	The task will have been removed from the entry queue
wait	abort	completed (abnormal)	The time out will have been canceled
	time_out	active	

Table 4.2.2 - Task State Transitions (continued)

*The status of a task is never actually set to complete_block, as no other task needs this information.

longer term view of task status transitions, where these equivalent states have been collapsed.

Another observation that can be made about table 4.2.2 is that there are event conflicts for several statuses. Fortunately, all of these events cannot occur concurrently. For example, even though it is possible that either an *entry_call* or a *terminate* event happens to a task waiting on a terminate alternative, the rules for task termination ensure that *both* an *entry_call* and a *terminate* event will not happen (see §4.4). Other event conflicts are possible

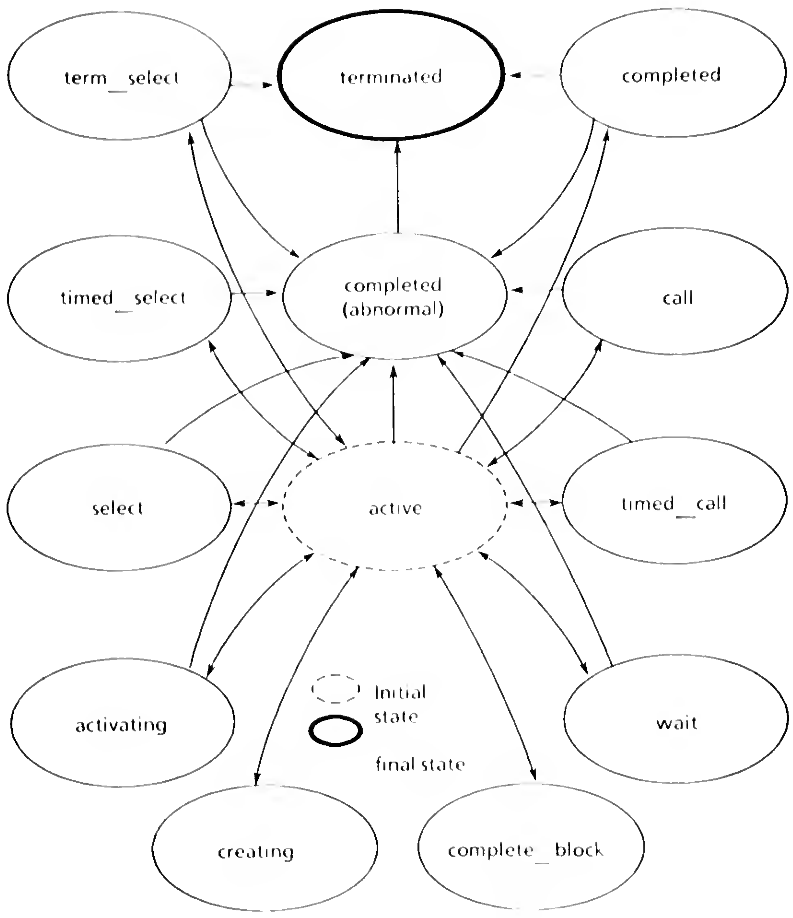


Figure 4.2.1: Task Status Transitions

– these conflicts must be resolved so that only one event occurs, and hence, causes a blocked task to be unblock. (The other conflicting events must be postponed.)

The possible event conflicts are: a) *abort*, *end_rdv* and *time_out*, b) *abort*, *start_rdv* and *time_out* and c) *abort* and *terminate* (only when a task is waiting at a select with a

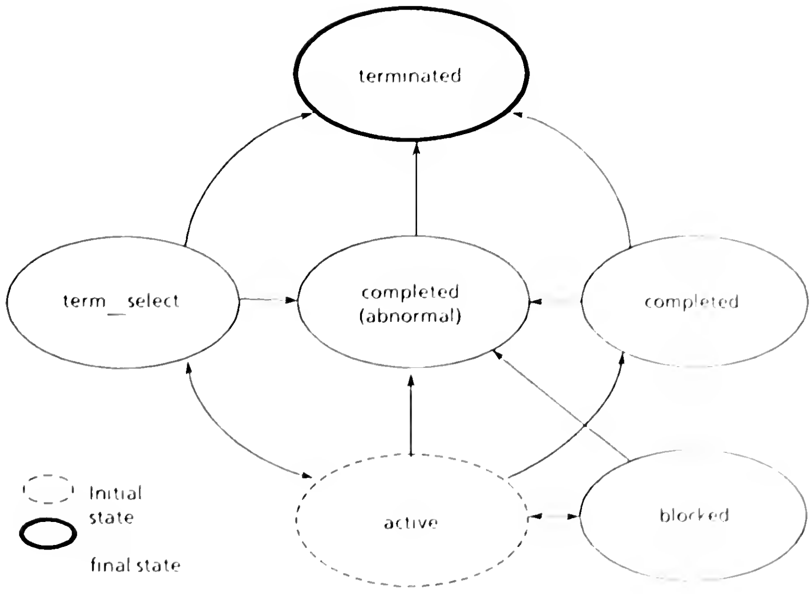


Figure 4 2 2 Task Status Transitions (Reduced)

terminate alternative) The resolution of *a*) and *b*) is discussed in detail in §4 5, §4 6, and §4 7 briefly, each event must disable the others for it to succeed The resolution of *c*) is discussed in detail in §4 4 and §4 7 the *abnormal* flag of the *TCB* of a task is used to resolve this event conflict

We ensure that it is also impossible for a second event to happen before the task become unblocked and inspects its event field, by requiring that the task itself enable its next event Below we list the actions that a task must perform to enable each event

abort The task must set its *status* to one of *calling*, *select*, *term _ select*, *timed _ call* or *timed _ select* see §4 7

activated :	The parent task must place the tasks to be activated on a ready queue - see §4.3.
created :	The parent task must place the tasks to be created on a ready queue - see §4.3.
done :	The master task must decrement the *num__deps* counter of one of its nested blocks - see §4.4.
end__rdv :	The calling task must unblock the owner of the called entry with a *start__rdv* event or place itself on the entry queue - see §4.6.
start__rdv :	The owner task must set its *rdv* flag - see §4.6.
terminate :	The master task must decrement its *no__term* counter - see §4.4.
time__out :	The delaying task must place the *time__node* pointed to by its *time__node__ptr* on a time chain - see §4.5.

4.2.3 The Scheduler

In this section we describe the implementation of the kernel of SMARTS. Specifically, we describe how the SMARTS processors are initialized with Ada tasks, how the Ada tasks are subsequently scheduled on the SMARTS processors, and how the SMARTS processors terminate (after all Ada tasks have terminated). The two scheduling policies that we discuss are run-until-blocked and pre-emptive. A SMARTS processor will run an Ada task until the tasks blocks or the SMARTS processor is pre-empted. When a blocked task is unblocked, the task must be enqueued onto a ready queue. When a SMARTS processor is pre-empted, the task currently being run must be enqueued onto a ready queue.

4.2.3.1 SMARTS Initialization

SMARTS begins its execution on a single processor. It requests a user supplied number of processors *num_pe* from the operating system with the kernel function *reserve(num_pe)*. The ready queues (one for each priority) are allocated; *num_runnable* is set to *num_pe* + 1 and the main task is enqueued on the ready queue for its priority. A SMARTS processor is assigned to each allotted processor via the *spawn(num_pe*, SMARTS processor) kernel function.

The SMARTS processors will execute the subprogram *initialize_pe* which, after decrementing *num_runnable*, attempts to dequeue an Ada task from the ready queues in a round robin fashion starting with the ready queue of highest priority. If *num_runnable* drops to 0 before the SMARTS processor successfully obtains a ready task, then the SMARTS processor terminates – there are no active tasks.

If a SMARTS processor is successful in dequeuing a task *t*, then after waiting until *t.switched* is **true** it switches in *t*. *initialize_pe* is also used to re-initialize a SMARTS processor after the task that it is executing terminates.

The elements of the TCB used to save the context of a task are the *stack_ptr* and the *stack_position_ptr*. When a SMARTS processor stops executing a task, all of the information necessary to execute the task is pushed onto its stack (e.g. its instruction pointer) and its code and data are flushed from the cache (of the processor that is executing the SMARTS processor). When a SMARTS processor begins executing a new task this information is popped from the stack of the task. (See [Kruchten 1985] for a description of the actual execution of Ada tasks in SMARTS.)

4.2.3.2 Block Points

When a task *t* being executed by a SMARTS processor reaches a block point (i.e. a point in the execution of *t* where some outside action is necessary for *t* to continue) that enables a time out for *t* (i.e. a wait statement or a select with a delay statement), the subprogram *timed_block_task(t)* is called. *timed_block_task* checks if an event is pending for the task by decrementing *t.num_event* with a fetch&add(*t.num_event*, -1). If *t.num_event* is non-negative, then an event is pending, and control is returned to *t*.

If *t.num_event* was 0, then no event is pending and a context switch is performed. First, the task is switched out, the cache is flushed and *t.switched* is set to **true**. Setting *t.switched* to **true** signals that *t*'s private and synchronous data has been effectively stored in the global memory (see §4.8), and hence, *t* is free to be executed by another SMARTS processor. After this, an attempt is made to dequeue an Ada task from a ready queue as is done in *initialize_pe* (but, note that *num_runnable* is not decremented as *t* can be unblocked by a time out). If a task is found then, the SMARTS processor waits for the *switched* field of the task to be set to **true** (i.e., until the task has actually been switched out by its previous SMARTS processor). Finally, the task is switched in, completing the context switch.

When a task *t* being executed by a SMARTS processor reaches a block point, other than when it enables a time out or terminates, the subprogram *block_task(t)* is called. *block_task* behaves the same as *timed_block_task*, except that *num_runnable* must be decremented when there is no event pending for *t*, since *t* requires an active task to unblock it. When a task has a pending time out it can be made active by the timer interrupt handler. When a task terminates, *initialize_pe* is called to re-initialize the SMARTS processor. Recall that *initialize_pe* decrements *num_runnable*.

For situations where a task knows that some event is about to occur (but does not know which event), we provide a subroutine *busy_wait*. *busy_wait(t)* simply spins until *t.num_event* is non-zero and then decrements it. By having the task busy wait for the event a context switch is avoided. Busy waiting should not be used in conjunction with a pre-emptive scheduling policy, as the task generating the event could be pre-empted (and hence, be prevented from generating the event).

4.2.3.3 Unblocking Tasks

When an event *e* occurs that allows the resumption of a task *t* that was blocked at a point where a time out had been enabled for *t* (i.e., a wait statement or a select with a delay statement), the subprogram *timed_unblock_task(t, event)* is called. *timed_unblock_task* sets *t.event* to *e* and increments *t.num_event* with a fetch&add(*t.num_event*, 1). If *t.num_event* is now 0, then *t* is enqueued on the ready queue for its priority. Otherwise *t* is currently active, and hence, no action is required.

Before enqueuing *t* onto this ready queue interrupts must be software disabled by setting the *preempt_lock* flag of the SMARTS processor to 1. If the time slice of the SMARTS processor expires (and hence, the *timer_handler* is invoked) while software interrupts are disabled, then the SMARTS processor will not be immediately pre-empted. Instead the *timer_handler* will increment the *preempt_lock* flag of the SMARTS processor signaling that a pre-emption is pending (see §4.5.4). After *t* has been enqueued onto the ready queue, the *preempt_lock* is decremented by 1. If the *preempt_lock* is not now 0, then the pending pre-emption of the SMARTS processor is honored (as described below), and software interrupts are re-enabled by setting the *preempt_lock* to 0.

When an event *event* occurs that allows the resumption of a task *t* that was blocked at a point where a time out had not been enabled for *t*, the subprogram *unblock_task(t, event)* is called. *unblock_task* behaves the same as *timed_unblock_task* except that *num_runnable* must be incremented when *t.num_event* is enqueued onto a ready queue. (Note that *num__runnable* is also incremented when a task is created.)

There are two other versions of *timed__unblock__task* and *unblock__tasks* that are used by the SMARTS routines that implement interrupt entries calls. (Actual calls to interrupt entries are issued by hardware devices. These routines perform the actions necessary to engage in a rendezvous with the owner of the interrupt entry when an interrupt is generated by the device. See §4.6.) *timed__unblock__and__run* behaves like *timed__unblock__task* except that it attempts to switch in the task being unblocked if the task is not currently running (i.e., is blocked). Analogously, *unblock__and__run* behaves like *unblock__task* except that it attempts to switch in the task being unblocked if the task is not currently running.

4.2.3.4 Pre-emption of SMARTS Processors

When a time slice expires on a processor, the timer interrupt handler is invoked. After timing out any tasks whose delays have expired, the timer interrupt handler will determine if the SMARTS processors should be pre-empted, i.e., if the current tasks's time slice is up. The *timer__handler* will not attempt to pre-empt the SMARTS processor during a ready dequeue or enqueue operation. In the former case, the SMARTS processor is already attempting to perform a context switch. In the later case, the task will voluntarily relinquish the SMARTS processor when the the ready enqueue operation is complete. (See §4.5.4 for details on the implementation of the *timer__handler*.)

Before and if so, will perform a task context switch (as described above). (See §4.5 for details on the implementation of the timer interrupt handler.) If the task *t* being run by the SMARTS processors has priority *p*, then the handler must check if the SMARTS processor should be pre-empted, i.e., if there is a ready-to-run task of priority higher than *p*. Thus, the handler calls the subroutine *preempt__pe* which searches the ready queues from $p + 1$ to MAX__PRIORITY. (Recall that there is a ready queue per priority level.) If such a task is found, then *t* is switched out and enqueued onto the *p*th ready queue and that task is switched in.

Before entering a critical section protected by an A/B-lock (e.g., during a ready queue operation) a task must calls the kernel function *mask_interrupts* which execute a machine instruction for masking interrupts. While interrupts are masked the task cannot be preempted as a timer interrupt cannot occur. (Note that if interrupt entries are not used then only timer interrupts need to be masked during the critical section. See §4.5 for a discussion of interrupt entries.) When the critical section is left, interrupts are unmasked by calling the kernel function *unmask_interrupts*

4.2.4 Code for Scheduling

-- Code for Kernel Initialization.

```
procedure initialize_pe is
-- Initialize a SMARTS processor (either it was idle or the task it was executing
terminated).
begin
    -- Software disable preemption, as we are already context switching.
    PREEMPT_LOCK := 1;
    -- If there are runnable tasks...
    if fetch_and_add(NUM_RUNNABLE, -1) > 0 then
        -- try to get one.
        TASK := get_task;
        if TASK /= null then
            switch_in(TASK); -- We are now executing TASK.
            PREEMPT_LOCK := 0; -- Re-enable preemption, as we have a new task.
            return;
        end if;
    end if;
    -- It's all over, so disable any pending timer interrupts.
    stop_timer(VALUE);
    -- Terminate ...
end initialize_pe;
```

-- Code for Searching Ready Queues

```
function get_task return TCB_PTR is
-- Get a ready to run task (if there is one).
begin
```

```
    loop -- Until we get a ready-to-run task or SMARTS terminates.
        -- Get the ready task with highest priority.
        for PRIORITY in reverse 1..MAX_PRIORITY loop
            TASK := ready_dequeue(READY_QUEUE(PRIORITY));
            if TASK /= null then -- We found a ready to run task.
                return TASK;
            end if;
            -- Return null when there are no runnable tasks.
            if read(NUM_RUNNABLE) = 0 then
                return null;
            end if;
            -- If we are pre-empted, then start over with highest priority queue.
            if PREEMPT_LOCK /= 1 then
                PREEMPT_LOCK := 1;
                exit; -- Exit for loop.
            end if;
        end loop;
    end loop;
end get_task;
```

-- Code for Ready Queue Operations.

```
procedure ready_enqueue(N; Q) is
-- Enqueue a task onto the ready queue Q.
begin
    PREVIOUS := fetch_and_store(Q.LAST, N);  -- Make N last on queue.
    if PREVIOUS = null then   -- The queue was empty, update first.
        write(Q.FIRST, N);
        write(Q.FIRST_MULT, N.MULT); -- Dequeues enabled.
    else  -- Insert the node into the queue
        write(PREVIOUS.NEXT, N);
    end if;
end ready_enqueue;

procedure ready_enqueue_list(FIRST_N, LAST_N;  Q) is
-- Enqueue the list of tasks starting with FIRST_N and ending with LAST_N onto
-- the ready queue Q.
begin
    PREVIOUS := fetch_and_store(Q.LAST, LAST_N);  -- Make LAST_N last on queue.
    -- Insert the list onto the queue.
    if PREVIOUS = null then   -- The queue was empty, update first.
        write(Q.FIRST, FIRST_N);
        write(Q.FIRSTMULT, FIRST_N.MULT); -- Dequeues enabled.
    else  -- Insert the list into the queue.
        write(PREVIOUS.NEXT, FIRST_N);
    end if;
end ready_enqueue_list;

function ready_dequeue(Q) return TCB_PTR is
-- Dequeue a task from the ready queue Q.
begin
    while read(Q.LAST) /= null loop
    -- There are nodes to be had...
```

```
        if (read(Q.FIRSTMULT) > 0) and then (fetch_and_add(Q.FIRSTMULT, -1) > 0) then
            -- We have reserved an item in the first node.
            OLD_FIRST := read(Q.FIRST);
            -- We may need I later to index into the TCB_PTRS array.
            I := fetch_and_add(OLD_FIRST.MULT, -1);
            if I = 1 then
            -- We are the last to dequeue an item, so get a new one.
                write(Q.FIRST, OLD_FIRST.NEXT);
                if read(Q.FIRST) = null then
                -- The queue is empty.
                    OLD_LAST := fetch_and_store(Q.LAST, null);
                    -- Note, that dequeues can now be enabled by an enqueuer.
                    if OLD_FIRST /= OLD_LAST then
                    -- Some nodes were enqueued, so enqueue to end of queue.
                        while read(OLD_FIRST.NEXT) = null loop
                        -- Wait until the first of the nodes is fully enqueued.
                        end loop;
                        enqueue_list(OLD_FIRST.NEXT, OLO_LAST, Q);
                    end if;
                else
                    -- The queue is not empty so dequeue.
                    -- So enable dequeues..
                    write(Q.FIRSTMULT, Q.FIRST.MULT);
                end if;
            end if;
            -- Determine the appropriate action for the node.
            case read(OLD_FIRST.STATUS) of
            when INITIALIZE =>                       -- Initialize TCB of multi-item.
                                                     -- And then get another task.
                initialize(OLD_FIRST.TCB_PTRS(1), I, OLD_FIRST);

            when ACTIVATE =>                         -- Activate TCB of multi-item.
                return OLD_FIRST.TCB_PTRS(I);        -- Get appropriate TCB.

            when TERMINATE =>                        -- Terminate TCB of multi-item.
                TASK := OLD_FIRST.TCB_PTRS(I);       -- Get appropriate TCB.
                if fetch_and_store(TASK.RDV, false) = true
                -- Waiting on a terminate alternative,
                and then fetch_and_add(TASK.NUM_EVENTS, 1) = 0 then
                -- and blocked.
                    return TASK;
                else
                    -- NUM_RUNNABLE was incremented for each TCB_PTRS, and this
                    -- one does not need to be unblocked, so adjust NUM_RUNNABLE.
                    add(NUM_RUNNABLE, -1);
                end if,
            when others =>                           -- Single item, return TCB
                return OLD_FIRST;
            end case.
    end loop
    return null; -- The queue is empty.
end ready dequeue;
```

-- Code for Context Switching.

```
procedure switch_out(TASK) is
-- Perform the first half of a context switch, i.e., switch a task out.
begin
    push(PC); -- Save state information.
    ...
    push(IC);
    flush(all); -- Save local data.
    write(TASK.SWITCHED, true); -- Signal that task is switched out.
end switch_out;

procedure switch_in(TASK) is
-- Perform the second half of a context switch, i.e., switch a task in.
begin
    -- Wait until state is saved.
    while not fetch_and_store(TASK.SWITCHED, false) loop
    end loop;
    pop(IC); -- Get state information.
    ...
    pop(PC);
    TASK_PTR := TASK;
end switch_in;
```

-- Code for Block Points.

```
procedure timed_block_task(TASK) is
-- TASK is blocked in a wait state, so if no events are pending for TASK, begin
-- executing another task.  Since TASK is waiting on a time out, it is still active.
begin
    if fetch_and_add(TASK.NUM_EVENT, -1) > 0 then   -- An event is panding.
        return;
    else
        -- Software disable preemption, as we are already context switching.
        PREEMPT_LOCK := 1;
        -- No event pending, so switch out task.
        switch_out(TASK);
        -- Try to get another
        NEW_TASK := get_task;
        if NEW_TASK /= null then
            switch_in(NEW_TASK); -- We are now executing NEW_TASK.
            PREEMPT_LOCK := 0; -- Re-enable preemption, as we have a new task.
            return;
        end if;
        -- It's all over, so disable any pending timer interrupts.
        stop_timer(VALUE);
        -- Terminate
    end if;
end timed_block_task;
```

```
procedure block_task(TASK) is
-- TASK is blocked in a non-wait state, so if no events are pending for TASK, begin
-- executing another task.
begin
    if fetch_and_add(TASK.NUM_EVENT, -1) > 0 then    -- An event is panding.
        return;
    else
        -- Software disable preemption, as we are already context switching.
        PREEMPT_LOCK := 1;
        -- No event pending, so switch out task.
        switch_out(TASK);
        -- If Num_runnable tasks still exist, try to get one.
        if fetch_and_add(NUM_RUNNABLE, -1) /= 1 then
            NEW_TASK := get_task;
            if NEW_TASK /= null then
                switch_in(NEW_TASK); -- We are now executing NEW_TASK.
                -- Re-enable preemption, as we have a new task.
                PREEMPT_LOCK := 0;
                return;
            end if;
        end if;
        -- It's all over, so disable any pending timer interrupts.
        stop_timer(VALUE);
        -- Terminate ...
    end if;
end block_task;

procedure busy_wait(TASK) is
-- Busy-wait for an event that we are certain is in progress.
-- This routine should not be used in conjunction with pre-emptive scheduling.
begin
    while (TASK.NUM_EVENT = 0) loop
        null;
    end loop;
    add(TASK.NUM_EVENT, -1);
end busy_wait;
```

-- Code for Unblocking Tasks.

```
procedure timed_unblock_task(TASK, EVENT) is
-- Set TASK's event and then make  TASK ready to run (if it is not already ready).
-- Since TASK is waiting on a time out it still Num_runnable.
begin
    write(TASK.EVENT, EVENT);
    if fetch_and_add(TASK.NUM_EVENT, 1) = -1 then
    -- TASK is not currently running, so add it to the ready queue.
        PREEMPT_LOCK := 1;  -- Software disable interrupts.
        ready_enqueue(TASK, READY_QUEUE(TASK.PRIORITY));
        if fetch_and_add(PREEMPT_LOCK, -1) /= 1 then
        -- Will software enable interrupts only if we were not pre empted
            preempt_pe;  -- We were preempted
            PREEMPT_LOCK := 0; -  Software enable preemption
        end if;
        end if;
    end timed_unblock_task;

    procedure unblock_task(TASK, EVENT) is
    -- Set TASK's event and then make  TASK ready to run (if it is not already ready).
    begin
        write(TASK.EVENT, EVENT);
        if fetch_and_add(TASK.NUM_EVENT, 1) = -1 then
        -- TASK is not currently running, so add it to the ready queue.
        -- One more ready to run task (Note Num_runnable > 0, as this is being executed
        -- by an active task).
            add(NUM_RUNNABLE, 1);
            PREEMPT_LOCK := 1; -- Software disable interrupts.
            ready_enqueue(TASK, READY_QUEUE(TASK.PRIORITY));
            if fetch_and_add(PREEMPT_LOCK, -1) /= 1 then
            -- Will software enable interrupts only if we were not pre-empted.
                preempt_pe; -- We were preempted.
                PREEMPT_LOCK := 0; -- Software enable preemption.
            end if;
        end if;
    end unblock_task;
```

-- Code for Unblocking Device Processes.

```
    procedure timed_unblock_and_run(TASK, EVENT) is
    -- Unblock a task that is waiting on a delay when one of its interrupt entries
    -- has been called.
    -- If the task is blocked (and we have not interrupted a device process)
    -- start running the task.
    begin
        write(TASK.EVENT, EVENT);
        if fetch_and_add(TASK.NUM_EVENT, 1) = -1 then
        -- The task is not currently running.
        -- Note, preemption is already disabled.
            if read(TASK_PTR.PRIORITY) /= MAX_PRIORITY
            and then fetch_and_add(PREEMPT_LOCK, 1) = 0 then
            -- Context switch.
                switch_out(TASK_PTR);
                switch_in(TASK);
                PREEMPT_LOCK := 0;
            else
            -- We interrupted a device process, or context switch.
            -- Note that as the ready enqueue does not busy wait, this cannot
            -- lead to a deadlock with the interrupted task.
                ready_enqueue(TASK, READY_QUEUE(MAX_PRIORITY));
            end if;
        end if;
    end timed_unblock_and_run;

    procedure unblock_and_run(TASK, EVENT) is
    -- Unblock a task when one of its interrupt entries has been called.
    -- If the task is blocked (and we have not interrupted a device process)
    -- start running the task.
```

```
begin
    write(TASK.EVENT, EVENT);
    if fetch_and_add(TASK.NUM_EVENT, 1) = -1 then
    -- The task is not currently running.
        add(NUM_RUNNABLE, 1); -- One more active task.
        -- Note, hardware preemption is already disabled.
        if read(TASK_PTR.PRIORITY) /= MAX_PRIORITY
        and then fetch_and_add(PREEMPT_LOCK, 1) = 0 then
        -- Context switch.
            switch_out(TASK_PTR);
            switch_in(TASK);
            PREEMPT_LOCK := 0; -- Software enable interrupts.
        else
        -- We interrupted a device process, or context switch.
        -- Note that as the ready enqueue does not busy wait, this cannot
        -- lead to a deadlock with the interrupted task.
            ready_enqueue(TASK, READY_QUEUE(MAX_PRIORITY));
        end if;
    end if;
end unblock_and_run;
```

-- Code for Pre-empting SMARTS processors.

```
procedure preempt_pe is
-- Called when a time slice expires, to see if SMARTS processor should be pre-empted.
begin
    -- See if there are ready to run tasks of higher priority than TASK.
    for PRIORITY in reverse read(TASK_PTR.PRIORITY) + 1..MAX_PRIORITY loop
        NEW_TASK := ready_dequeue(READY_QUEUE(PRIORITY));
        if NEW_TASK /= null then -- We found a ready to run task.
            exit;
        end if;
    end loop;
    if NEW_TASK /= null then
    -- We found a new task, so switch out old.
        switch_out(TASK_PTR);
        -- Enqueue the task as it is not blocked.
        ready_enqueue(TASK_PTR, READY_QUEUE(TASK.PRIORITY));
        switch_in(NEW_TASK); -- We are now executing NEW_TASK.
    end if;
end preempt_pe;
```

4.3 Task Creation and Activation

The initial phase of execution where a task elaborates the declarative part of its task body is called the *activation* of the task. The task whose execution causes a task *p* to be activated is said to be the parent of *p*. The Ada language is very specific about when tasks are created and when tasks begin execution. In particular, the language specifies what tasks must be created and activated in parallel.

A task object whose declaration appears immediately within a declarative part is created when the object declaration for the task is elaborated. Such task objects are activated after the elaboration of the declarative part that declares the task object. Thus, task objects, though they may be created sequentially, are activated in parallel with the other task objects whose declaration appear within the same declarative part (context). The first statement of the body of the parent is not executed until all of the tasks objects whose declaration appear within the declarative part have completed their activation.

Although activation is defined as the initial phase of execution of a task, a task must be activated at the maximum priority level of the task and its parent [AI-00288]. If only one of the task and its parent specifies a priority, then the activation is executed with at least that priority. Allowing a child task to be activated at the priority of its parent can avoid priority inversion. (Recall that priority inversion is a situation where a task of lower priority blocks a task of higher priority.) To wit, consider a parent task *p* with priority level 5, that is blocked awaiting the activation of its child task c with priority level 3. If c must activate at priority level 3, then a task *t* with priority level 4 could prevent c from executing, thereby, causing *p* (which has a higher priority level than *t*) to remain blocked.

If an exception is raised during the activation of a task, that task becomes completed. When some of the tasks that are being activated by a task *p* become completed during activation because of an exception, the exception TASKING_ERROR is raised in *p* (If more than one task becomes completed because of an exception then TASKING_ERROR is raised only once.). When an exception is raised during the elaboration of a declarative part of a task or a package, any tasks created during this elaboration that are not yet activated become terminated. (Similarly, when a task is aborted, any tasks created by the task and that are not yet activated become terminated.)

A task object that is created by the evaluation of an allocator is activated by this evaluation. If more than one task is created by the evaluation of an allocator these tasks are activated in parallel. The following example from [Rosen 1985] demonstrates the difference between the creation and activation of: *a*) tasks which are created by allocators and *b*) tasks which are created by object declarations.

```
procedure main is
task type t;
type access_t is access t;
type array_access_tasks is array(1..10) of access_t;
type array_tasks is array(1..10) of t;
type access_array_tasks is access array_tasks;

A1 : array_tasks; -- Create 10 tasks.

 -- Create A2(1) and activate A2(1), then create A2(2) and activate A2(2), then ...
A2 : array_access_tasks := (others => new t);

-- Create 10 tasks and then activate them in parallel.
A3 : access_array_tasks := new array_tasks;
begin  -- Activate the 10 tasks of A1.
    ...
end;
```

Since the programmer has control over which tasks can be created and activated in parallel, we use this information to guide our implementation.

SMARTS uses the same mechanism as Ada/ED to determine when families of tasks are to be created and activated (tasks with the same parent and master). For completeness, we will briefly describe this mechanism here. For details, consult [Rosen 1985]. The tasks that are to be activated together are linked onto the same *task frame*. Task frames that contain lists of tasks that have been created, but not yet activated are linked together to from a stack. After the tasks linked onto a task frame have been activated, the task frame is popped from the stack. Thus, a task frame consists of a pointer to a task list and a pointer to the previous task frame. The *task__declared* pointer of the current block frame (*BF*) points to the top of the stack, that is, the most recently created task frame. (*BF*'s are defined in the next subsection.) After the tasks pointed to by the *task__declared__ptr are* activated, the top of the stack is popped, i.e., *task__declared* is set to *task__declared.previous__task__frame.* If a new task frame *ntf* is created before the task frame pointed to by *task__declared* is activated, then it is pushed onto the stack, i.e., *ntf.previous__task__frame* is set to *task__declared* and *task__declared* is set to *ntf.* In the next two subsections, we describe in detail how (as opposed to when) tasks are actually created and activated in a highly-parallel manner by SMARTS.

The fields of *TCB*'s and *BF*'s that are involved in the creation and activation of tasks are discussed in §4.3.2. Before discussing these data structures, we define the elements that make up a *BF*. *TCB*'s are defined in §4.1.

4.3.1 Block Frames

BF's contain the fields necessary to coordinate the dependents of a master block: *first__dep, last__dep, no__term, num__deps* and *tasks__declared*. Note that, like *TCB's*, *BF*'s have a *no__term* field for terminating dependent tasks (see §4.6). *first__dep* and *last__dep* are used to manage a queue of the direct dependents of the master block. (For compatibility with ready queues, the queue is implemented with the algorithm described in §3.2.2.3.) *Task__declared* points to the current set of tasks that have been created (but not yet activated) in the same context within the block (i.e. they have the same parent and master block); these tasks will be activated together. Table 4.3.1 lists the elements of a *BF*.

Element	Use
block__ptr	Pointer to the enclosing block frame.
data__ptr	Pointer to the data segment of the block.
exception__vector	Points to the current exception handler.
first__dep	Head of the dependents chain (Linked by sibling pointers.).
last__dep	Tail of the dependents chain (Linked by sibling pointers.)
no_term	The number of nonterminatable direct dependents.
num_deps	The number of nonterminated direct dependents.
tasks__declared	Set of tasks to be created and activated together at end of the elaboration of the declarative part of the block.

Table 4.3.1 - Block Frame

4.3.2 Data Structures for Task Creation and Activation

To create and activate tasks SMARTS makes use of the following fields of the *TCB*: *exception*, *num__event*, *priority*, *save__priority*, *status* and *who*. The fields of the *TCB* needed to complete and terminate tasks, *master__task*, *master__block* and *num__term* (see §4.4), must also be set when the task is created. Further, since the point where a task causes the activation of another task and the end of the activation of a task are both abort synchronization points (see §4.8), the *abnormal* flag of a *TCB* is peripherally involved in the activation of tasks.

The *who* field of the *TCB* of a task is set to the *TCB* of its parent (activator) when the task is created (or initialized). (Since after being activated, a task has no interactions with its parent, the *who* field will later be used to point to the tasks that called an entry of the task.)

When a set of tasks is being initialized or activated in parallel, the *num__event* field of the *TCB* of their parent is used to determine when the initialization or activation is completed. If the set contains m tasks, then *num__event* is set to $-m+1$, signifying that there are m

outstanding initialization or activation events (the extra one is to compensate for the parent decrementing *num__event* when it blocks).

The *priority* and *save__priority* are used to execute the activation of the task at a priority equal to either that of the task or its parent, whichever is higher.

A task is a dependent of its master task and master block. This dependency relationship is used to determine which sets of tasks are terminated together (see §4.4). The *master__task* field of the *TCB* of a task points to the *TCB* of its master task. The *master__block* field of the *TCB* of a task points to the *BF* of its master block. The *num__deps* and *no__term* fields of the *BF*'s of *master__block*'s and the *no__term* field of the *TCB*'s of *master__task*'s are used to implement task termination and completion. They must be incremented when a dependent task is activated. (The *no__term* and *num__deps* of a *master__block* and the *no__term* of a *master__task* are all initially 1, signifying that the task itself is active.)

The *first__deps*, *last__deps* and *tasks__declared* of the *BF* of the *master__block* of a task are also used in the creation and activation of the task. *first__deps* and *last __deps* point to the first and last direct dependents of the block (respectively). *task__declared* points to the current set of tasks that have been created, but not yet activated, in the same context (i.e. they have the same parent and master block) within the block; these tasks will be activated together.

The Ada language specifies when tasks may be activated in parallel. In order to avoid serial bottlenecks in SMARTS we also perform other functions, e.g., initialization and termination, on arrays of tasks in parallel. When a task *p* elaborates an array, *p* creates a *SMARTS__node* with multiplicity (equal to the number of tasks in the array). When it is necessary to perform certain operations on the array of tasks this *SMARTS__node* will be enqueued on a ready queue, so that the operation will (potentially) be performed on the array of tasks in parallel by the SMARTS processors. Later in this section, we discuss the parallel initialization and activation of arrays of tasks.

SMARTS_nodes and their scheduling are described in §4.2 The *SMARTS_node* fields used specifically for task initialization are the two pointers: *parent* and *template*. *parent* points the activator of the array of tasks. *template* points to the task type template of the array of tasks. The type template contains all the structural information necessary to create the tasks.

4.3.3 Actions of the Creator and Activator of a Task

When a task of type *tt* is created serially by a task *p*, *p* allocates storage for the *TCB* of the task and then initializes the fields of this *TCB* with the appropriate information from *tt*. Note that a task must be placed on the dependent queue of its master *before* it is activated. This is to ensure that task is not missed by the SMARTS abort routine (which iterates over the dependent task list of an aborted task) when the master of the task is aborted. (Recall that when a master task is aborted its dependent tasks that are created, but not yet activated, are required to terminate immediately.)

Another complication arises from the requirement that a task is activated at the maximum of its own and its parent's priority. Thus, the priority specified by template is not copied to the *priority* field of the *TCB* of the task being created Rather, it is copied to the *save_priority* field of the *TCB*. The *priority* field of the *TCB* is set to the maximum of the priority specified by the template and the priority of the parent of the task.

When *m* tasks of type *tt* are initialized in parallel (i e. an array of tasks) by a task *p*, *p* allocates storage for their *TCB*'s and creates a *SMARTS_node* with multiplicity *m*. The *status* is set to *create*, and the *template_ptr* is set to *tt* Then, after setting its *num_event* count to $-m + 1$, *p* enqueues the *SMARTS_node* on the ready multi-queue for the priority of p. Thus the *TCB*'s of the tasks will be initialized in parallel by the SMARTS processors. p then blocks. Each SMARTS processor that initializes one of the *TCB*'s will attempt to unblock p (see §4.2.4). Since, *num_event* was set to $-m + 1$, only the last SMARTS processor will succeed. When p is unblocked, p checks if an exception has occurred. If an exception has not occurred then p sets the *status* of the *SMARTS_node* to *activate* (so that the tasks will later be activated in parallel). The *SMARTS_node* is then placed on the dependent queue of the *master_block* of the array of the tasks.

When a list of tasks (pointed at by *task_declared_ptr*) are to be activated together by a task p, p iterates through the list calculating the sum of their *mult* fields m and their maximum priority *mp*. (Before iterating through the list *mp* is initialized to the priority level of p, so that *mp* will be the maximum priority level of the tasks involved in the activation.) The tasks are also linked by their *next* pointers so that they can be enqueued as a list to a ready queue (the tasks will already be linked by their *sibling* pointers).

p then increments the *no_term* and *num_deps* counters of the *master_block* of the tasks by m, and also the *no_term* of the *master_task* of the tasks by m. (The rules for Ada ensure that all tasks which are activated together have the same master block and master task). After setting its *num_event* count to $-m + 1$ (the extra one is to compensates for *p.num_event* being decremented when p blocks), p enqueues the list of tasks (which consists of *TCB*'s or *SMARTS_node*'s) to be activated onto the appropriate ready queue. If the priority of p is undefined, then p can enqueue the entire list of tasks onto the ready queue for priority level *mp*. p then blocks. Otherwise, each *TCB or SMARTS_node* is enqueued to the ready queue for the maximum priority level of the *TCB or SMARTS_node* and p.

Each of the tasks being activated will try to unblock p after it is activated, again, only the last task will succeed. When p is unblocked, p checks if an exception has occurred, i.e., one of tasks was unable to activate. If an exception has occurred then p raises the exception (which, if unhandled, will cause all of the tasks that have been created by *p* but not yet activated to be terminated).

4.3.4 Actions of the Task being Initialized or Activated in Parallel

The initialization of the *TCB* of a task is straightforward. The *template__ptr* of the *SMARTS__node* points to the task type template of the task. The task type template contains most of the information necessary to initialize the *TCB*. The rest of the information can be found in the *SMARTS__node* itself.

The activation of a task consists of elaborating the declarations of the task. If the task is unsuccessful in elaborating its declarations, then the task sets the *exception* field of the *TCB* of its *parent* to TASKING__ERROR. Next the task attempts to unblock its parent. Finally, if its activation was successful, the task restores its *priority* level from its *save__priority* level and continues executing, otherwise the task completes.

4.3.5 Code for Task Creation and Activation

-- Code for the Creation of Tasks

```
procedure create_task(TASK; PARENT; TEMPLATE) is
--  PARENT creates a TASK of type TEMPLATE.
begin
    --  Create and initialize the task.
    TASK := new TCB_REC;
    push(TASK); -- Push the address of the TCB on the parent's activation stack.
    write(TASK.WHO, PARENT);
    write(TASK.MULT, 1);
    write(TASK.SAVE_PRIORITY, TEMPLATE.PRIORITY);
    MP := max(TEMPLATE.PRIORITY, TASK.SAVE_PRIORITY);
    write(TASK.PRIORITY, MP);
    write(TASK.MASTER_TASK, TEMPLATE.MASTER_TASK);
    write(TASK.MASTER_BLOCK, TEMPLATE.MASTER_BLOCK);
    write(TASK.NO_TERM, 1);
    write(TASK.NUM_DEPS, 1;
    write(TASK.STATUS, ACTIVATING). -- Will be when it's activated anyway
    add(NUM_RUNNABLE, 1); -- One more active task
    -- Etc.
       Add to master's dependents list
```

```
        deps_enqueue(TASK.MASTER_BLOCK, TASK);
        if read(TASK.MASTER_TASK.ABNORMAL) /= 0 then
        -- Newly created task will have been missed.
            terminated_unactivated(TASK);
        end if;
    end create_task;

    procedure create_tasks(PARENT; TEMPLATE) is
    -- PARENT is creating an array of tasks of type TEMPLATE.
    begin
        -- Create a SMARTS_NODE.
        S_NODE := new SMARTS_NODE_REC;
        write(S_NODE.STATUS, INITIALIZE);
        write(S_NODE.PARENT, PARENT);
        -- Allocate heap storage for the TCB's
        -- (will raise STORAGE_ERROR if not enough space).
        TASKS := new TCB_ARRAY(1..TEMPLATE.MULT);
        write(S_NODE.TCB_PTRS, TASKS);
        push(TASKS); -- Save start address of TCB's in PARENT's stack.
        -- Set parent MULT to number of tasks being initialized + 1.
        write(PARENT.NUM_EVENT, -TEMPLATE.MULT);
        fetch_and_add(ACTIVE, PARENT.MULT); -- MULT more active tasks
        -- Initialize the array at the priority of the parent.
        ready_enqueue(NODE, READY_QUEUE(PARENT.PRIORITY));
        -- Wait for tasks to be initiated.
        block_task(PARENT);
        -- Add to master block's dependents list.
        deps_enqueue(S_NODE, TEMPLATE.MASTER_BLOCK);
        if read(TEMPLATE.MASTER_TASK.ABNORMAL) /= 0 then
        -- New created tasks will have been missed.
            terminated_unactivated(S_NODE);
        else
            write(S_NODE.MULT, TEMPLATE.MULT);
            write(S_NODE.STATUS, ACTIVATE);
            write(S_NODE.PARENT, PARENT);
        end if;
    end create_tasks;
```

-- Code for the Initializer of Tasks

```
    procedure initialize_task(TASK; I; S_NODE) is
    -- Initialize one of an array of tasks from its SMARTS_node.
    begin
        PARENT := S_NODE.PARENT;
        TEMPLATE := read(S_NODE.TEMPLATE); -- Get task type template.
        -- Initialize the task.
        write(TASK.WHO, S_NODE.PARENT);
        write(TASK.MULT, 1);
        write(TASK.SAVE_PRIORITY, TEMPLATE.PRIORITY);
        MP := max(TEMPLATE.PRIORITY, TASK.SAVE_PRIORITY);
        write(TASK.PRIORITY, MP);
        write(TASK.MASTER_TASK, TEMPLATE.MASTER_TASK);
        write(TASK.MASTER_BLOCK, TEMPLATE.MASTER_BLOCK);
        write(TASK.NO_TERM, 1);
        write(TASK.NUM_DEPS, 1);
        write(TASK.STATUS, ACTIVATING);
        -- Etc. ...
        unblock_task(TASK.WHO, CREATED);
    end initialize_task;
```

-- Code for the Activation of Tasks.

```
    procedure activate_tasks(PARENT; MASTER;  BLOCK; TASK_LIST) is
    -- Activate the linked list of tasks with the same master task MASTER and block
    -- BLOCK.  (Note: only called with non-null TASK_LIST.)
    begin
        if write(PARENT.ABNORMAL) /= 0 then
            abortme(PARENT);
        end if;
        TASK := read(TASK_LIST);
        NUM := 0;
        MP := read(PARENT.PRIORITY);
        while TASK /= null loop
            NUM := NUM + read(TASK.MULT);
            if read(TASK.STATUS) = ACTIVATE then
                MP := max(TASK.TCB_PTRS(1).PRIORITY, MP)
            else
                MP := max(TASK.PRIORITY, MP);
            end if;
            write(TASK.NEXT := TASK.SIBLING)
            TASK := read(TASK.SIBLING);
        end loop;
        -- Update termination and completion information of master task and block.
        add(MASTER.NO_TERM, NUM);
        add(BLOCK.NO_TERM, NUM);
        add(BLOCK.NUM_DEPS, NUM);
        write(PARENT.MULT, -NUM); -- NUM outstanding activate events.
        if PRIORITY = UNDEFINED then
        -- All the tasks can be executed with same maximal priority.
            ready_enqueue(TASK_LIST, READY_QUEUE(MP));
        else
        -- Put tasks on queue for maximum of theirs and their parents priority.
            TASK := read(TASK_LIST);
            while TASK /= null loop
                if read(TASK.STATUS) = ACTIVATE then
                    ready_enqueue(TASK, READY_QUEUE(TASK.TCB_PTRS(1).PRIORITY));
                else
                    ready_enqueue(TASK, READY_QUEUE(TASK.PRIORITY));
                end if;
            end loop;
        end if;
        -- Wait for the tasks to activate.
        block_task(PARENT);
        if read(PARENT.EXCEPTION) = TASKING_ERROR then
            raise(TASKING_ERROR);
        end if
```

```
end activate_tasks;

procedure deps_enqueue(BLOCK; TASK) is
-- Enqueue TASK onto the DEPS chain of BLOCK (chained by sibling pointers).
begin
    PREVIOUS := fetch_and_store(BLOCK.LAST_DEP, TASK);
    if PREVIOUS = null then -- If the list was empty, then initialize FIRST_DEP.
        write(BLOCK.FIRST_DEP, TASK);
    else -- Insert onto end of list.
        write(PREVIOUS.SIBLING, TASK);
    end if;
end deps_enqueue;
```

-- Code for the Activator of a Task.

```
procedure activate(TASK) is
-- Activate a task.
begin
    if read(TASK.ABNORMAL) /= 0 then
    -- Do not activate a task that has been aborted.
        TERM_CODE := SUCCESSFUL;
    else
        -- Elaborate declarations.
    end if;

    -- If elaboration failed.
    if TERM_CODE = FAILED then
        -- Notify parent and die gracefully.
        write(TASK.PARENT.EXCEPTION, TASKING_ERROR);
        if fetch_and_add(TASK.PARENT.MULT, -1 ) = 2 then
            unblock_task(TASK.PARENT, NO_EVENT)
        end if;
        complete(TASK);
    else
        -- Notify parent and continue executing ...
        if fetch_and_add(TASK.PARENT.MULT, -1 ) = 2 then
            unblock_task(TASK.PARENT, NO_EVENT)
        end if;
    end if;
    if read(TASK.ABNORMAL) /= 0 then -- This is a synchronization point.
        abortme(TASK, NO_TERM);
    end if;
    write(TASK.PRIORITY, TASK.SAVE_PRIORITY);
end activate; -- Continue execution ...
```

-- Utility Routine.

```
procedure terminate_unactivated(TASK_LIST) is
-- Terminate tasks that have been created, but not yet activated when their parent
-- is abnormally terminated or has an unhandled exception raised.
-- Should we loop through all blocks??
```

```
begin
    while read(TASK_LIST) /= null loop
        if read(TASK_LIST.STATUS) = ACTIVATE then
        -- If a multi-item (next action was to be activation).
            -- MULT less active tasks.
            add(NUM_RUNNABLE, -read(TASK_LIST.MULT));
            for I in 1..read(TASK_LIST.MULT) loop -- Could be done in parallel.
                write(TASK_LIST.TCB_PTRS(I).STATUS, TERMINATED);
                -- Some may have already been called.
                purge_rdv(TASK_LIST.TCB_PTRS(I));
            end loop;
        else
            fetch_and_add(NUM_RUNNABLE, -1); -- One less active task.
            -- Ensure no other tasks call my entries.
            write(TASK_LIST.STATUS, TERMINATED);
            purge_rdv(TASK_LIST); -- Some may have already been called.
        end if;
        write(TASK_LIST,  TASK_LIST.SIBLING);
    end loop;
end terminate_unactivated;
```

4.4 Task Termination and Deallocation

Tasks termination imposes a synchronization point on several language constructs. In accord with the block structure discipline of Ada, the end of a block in which tasks have been declared is a synchronization point: execution of the block's outer context may not proceed until after all such tasks have terminated. Termination is complicated by the *terminate* alternative of select statements. Because of this feature, sets of task must be terminated together based on the state of all of the tasks in the set.

Many implementations in a single processor environment, therefore, rely on a centralized supervisor ([Rosen 1983], [Riccardi and Baker 1984]) to detect terminatability and to lock out other tasks during task termination synchronization. More complex solutions to task termination have been presented for distributed, nonshared memory machines, for example [Kafura 1985] and [Fisher and Weatherly 1986]. Bot of these solutions involve message passing among sets of tasks to achieve the effects of a centralized supervisor; for example, a task must sometimes exchange messages with its parent to determine whether to engage in a rendezvous. The solution to task termination presented in this section generalizes the single processor approach described in [Riccardi and Baker 1984] to multiprocessor environments by eliminating the centralized supervisor and distributing its work among the SMARTS processors. Our solution has previously been reported in [Flynn *et al.* 1988].

The simplicity of the distributed solution is surprising in view of the richness of Ada tasking features. The inclusion of task types and access to task types in ANSI/MIL-STD-1815 Ada, while providing an orthogonal and powerful language construct, seemingly complicates questions of distributed termination. For example, the claim of Clemmensen [Clemmensen 1982], that only the parent and siblings of a task waiting at a terminate alternative are able to call one of its entries, does not hold in ANSI/Ada Because task types behave like other types, tasks may be passed as parameters, thus an inner scope can pass task objects to procedures declared in an outer scope Additionally, a task object can be passed to another task via an entry call. Because of the existence of access task types, tasks can be dynamically created using an allocator, and pointers to tasks can be freely assigned to access task variables. Fortunately, Ada termination rules, described in the next section, allow a straightforward distributed implementation in spite of these complications.

4.4.1 Task Dependence and Termination

The full description of Ada task termination rules is given in ARM 9.4. We give a brief summary of these rules here, and illustrate them with a few examples.

Tasks are *directly dependent* on *masters*, where a master *m* is either another task, a block, a subprogram, or a library package. To precisely define direct dependency, we must consider two cases:

(1) a task *created by the evaluation of an allocator* depends directly *on the master that elaborates the corresponding access type definition;*

(2) otherwise, a task depends directly *on the master whose execution creates the task object.*

If the direct master of a task *t* is a block, then *t* is said to be an inner dependent of the task that is executing the block. Such a task, in effect, has two masters: its direct master, which is a block, will be referred to as its *master block*; an indirect master, which is the task executing the block, will be referred to as its *master task*. This distinction is essential to the algorithms presented below. If the direct master of a task *t* is another task, then *t* has a master task, but no master block. If the direct master of t is a subprogram, then the subprogram will also be referred to as the master block of *t*. In this case, *t* has a master block, but no master task.

For uniformity, we define the master task of a task *t* whose master block is a subprogram to be the task that declares the subprogram, and the master block of a task *t* whose direct master is a task to be the outermost block of the master task, thereby ensuring that all tasks have both a master task and a master block.

The notion of dependence is generalized as follows: a task t depends on a master block m if it directly depends on m, is an inner dependent of m, or depends on another master m' that depends on m.

Task termination then takes place under the following conditions. If a task has no dependent tasks it terminates when it has *completed*, i.e., when it has finished executing its statements. If a task has dependent tasks, it terminates when it has completed, *and* all of its dependents have terminated.

A block or subprogram which has dependent tasks can only be left when all of its dependents have terminated. For consistency we will say that a block or subprogram *terminates* when it is left.

This straightforward rule is complicated by the Ada terminate alternative, which provides another way in which a task may terminate. A task terminates when "*...its execution has reached an open terminate alternative in a select statement, and the following conditions are satisfied:*

- *The task depends on some master whose execution is completed (hence not a library package).*
- *Each task that depends on the master considered is either already terminated or similarly waiting on an open terminate alternative of a select statement.*

When both conditions are satisfied, the task considered becomes terminated, together with all tasks that depend on its master." We call the termination of a task and all of its dependents the *termination wave* of the task. Termination of the main program is subject to an additional special rule: the main program does not await the termination of tasks that depend on library packages. The language does not define whether tasks that depend on library are required to terminate

Finally, a task can test whether other tasks have terminated using the attribute TERMINATED which is defined as follows [ARM 9.9 (3)] :

T'TERMINATED *Yields the value TRUE if the task designated by T is terminated. Yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN.*

Some of the implications of these rules are demonstrated in the following examples. It is useful to represent the dependency relation among masters and dependent tasks by trees. The root node of a dependency tree is a master task or a master block, the children of a tree node are all the tasks that depend on the node. (A master block cannot be an interior node in the dependency trees, since a master block is not a dependent of any other construct. A task t that has both a master task and a master block is an interior node in two trees.) After each example, we give its dependency tree(s).

Example 1 - Tasks with Terminate Alternatives

This example illustrates nested tasks with terminate alternatives in select statements. A portion of the dependency tree from example 1 is shown in figure 4.4.1. If all of the tasks in the tree are executing their select loops, termination can take place only if all of the *children* and *grandchildren* tasks are waiting at terminate alternatives. All of the tasks in the tree will terminate together in a single termination wave. Note that a given *children* task may be called by any of the 15 *grandchildren* tasks, and that even if all of the *children* tasks are waiting at terminate alternatives, the state of one of the *children* tasks may change because another task (e.g., one of the *grandchildren* tasks) may invoke it, and so termination may not be possible.

```
task master;
task body  master  is

    task type child  is
```

```
        entry e1(i : integer);
    end task;

    children : array (1..5) of child;

    task body children is

        task type grandchild is
            entry e2(j : integer);
        end task;

        grandchildren : array (1..3) of grandchild;

        task body grandchild is
        begin -- Code executed by grandchildren.
            loop
                select
                    accept e2(J : integer) do
                        ...
                        children(J).e1(J);
                        ...
                    end e2;
                or
                    terminate;
                end select;
            end loop;
        end grandchild;

    begin -- Code executed by children.
        loop
            select
                accept e1(I : integer) do
                    ...
                end e2;
            or
                terminate;
            end select;
        end loop;
    end children;


begin
    -- Code executed by master.
end master;
```

Example 2 - Tasks that Exchange Tasks as Entry parameters

A task can also be called by a task that is not a dependent of its master, as illustrated below. The dependency tree for the program is shown in figure 4.4.2

```
task master;
task body master is
```

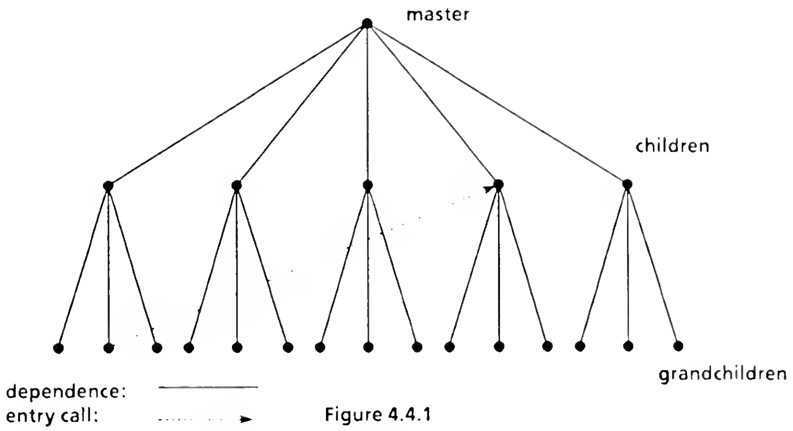


Figure 4.4.1

```
task middleman;

task type called_type is
    entry e1(J : in integer);
end called_type;

task caller is
    entry e2(tasktocall : in called_type);
end caller;

task body middleman is
    called : called_type;
begin
    ...
    caller.e2(called);
    ...
end;

task body called_type is
begin
    loop
        select
            accept e1(J : integer) do
                ...
            end e1;
        or
            terminate;
        end select;
        ...
    end loop;
end called_type;
```

```
    task body caller is
    begin
        ...
        accept e2(tasktocall : in called_type) do
            tasktocall.e1(17);
            ...
        end e2;
        ...
    end caller;

begin
    -- Code for master.
end master
```

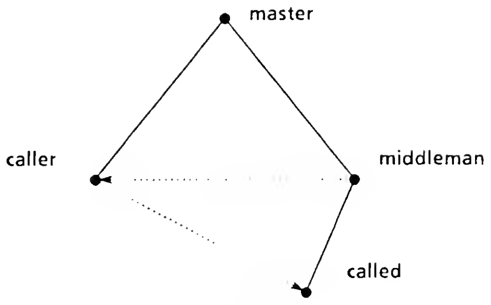


Figure 4.4.2

Example 3 - Tasks with Allocators

This example illustrates dynamic task allocation. The dependency tree is shown in figure 4.4.3. After the task *dependent* is activated, it dynamically creates tasks *t1* and *t2*. These dynamically created tasks are dependents of *dependent* because their access type is defined in dependent. Additionally, every time the entry *e* is called and the rendezvous is complete (e.g. see the body of master), *dependent* creates a new task *t3*, also a direct dependent of *dependent*. Termination can take place when *dependent* is waiting at a terminate alternative, all tasks created by dynamic allocators have terminated, and *master* is complete. Note that, depending on how many times the entry *e* is called, an arbitrary number of tasks *t3* may have been generated

```
task master;
task body master is

    task dependent is
        entry e(I : integer);
    end dependent;

    task type dynamic;
    task body dynamic is
    begin
        ...
    end dynamic;

    task body dependent is
        type dynamic_ptr is access dynamic;
        t1, t2, t3 : dynamic_ptr;

    begin
        t1 := new dynamic;
        t2 := new dynamic;
        loop
            select
                accept e(I : integer) do
                    ...
                    t3 := new dynamic;
                    ...
                end e;
            or
                terminate;
            end select;t2 := new dynamic;
        end loop;
    end dependent;

begin -- Body of master.
    dependent.e(7);
end master;
```

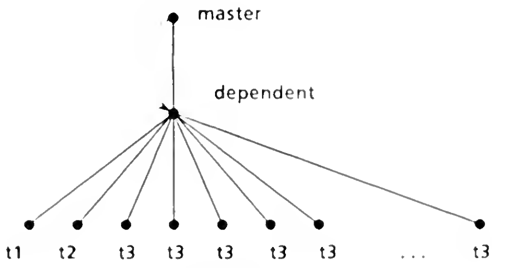


Figure 4.4.3

Example 4 - Master Blocks

This example illustrates tasks dependent on master blocks, and some of the complications that derive from dependency rules. Figure 4.4.4 shows the dependency trees of the master task and the master block. The declaration of task *t1* is elaborated by task *master*, so *t1* is a direct dependent of *master*. *master* contains an inner block: since the declaration of task *t2* is elaborated in the block, the block is the direct master of *t2*. Task *t2* is an inner dependent of master. Task *t3* is also created via an allocator within the inner block. However, since the access type of *t3* is declared in *master*, (see rule (1), §4.4.1), *t3* is the direct dependent of *master*.

The inner block can exit when its dependent *t2* terminates or waits on a terminate alternative and the block completes. Note, however, that the block itself contains a select loop with a terminate alternative. If the select loop is executed, it is possible that the inner block may never complete. In this case the task *master* (not the block) is waiting to terminate: master may then terminate according to the rules defined above, and when the direct and inner dependents of *master*, namely *t1*, *t2*, and *t3*, have terminated or are waiting to terminate. For this reason, task *t2* is in the dependency trees of both master and the inner block.

```
task master is
    entry em;
end master;

task body master is

    task type dependent is
        entry e;
    end dependent;

    type dependent_ptr is access dependent;

    task body dependent is
    begin
        loop
            select
                accept e do
                    ...
                end e;
            or
                terminate;
            end select;
        end loop;
end dependent;

t1 : dependent; -- direct dependent of master.

begin -- Body of master.
    ...
inner: declare
        t2 : dependent; -- Direct dependent of inner.
        t3 : dependent_ptr;
    begin
        ...
        t3 := new dependent; -- Direct dependent of master.
        ...
        loop
            select
                accept em do
                    ...
                end em;
            or
                terminate;
            end select;
        end loop;
        ...
    end inner;
end master;
```

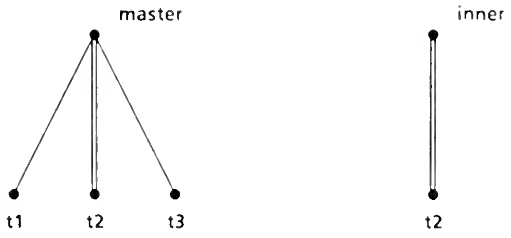


Figure 4.4.4

inner dependence: ═══════

4.4.2 A Necessary and Sufficient Condition for Termination

We introduce the notion of *quiescence*, and then prove a condition on dependency trees that enables a distributed implementation of termination.

Definition: If *t* is a task with no direct dependents, then *t* is *quiescent* if *t* is complete, or if *t* is waiting on a terminate alternative. If *t* has direct dependents, then *t* is *quiescent* if *t* is complete or if *t* is waiting on a terminate alternative, *and* all direct dependents of *t* are *quiescent*. A task which is neither quiescent nor terminated is said to be *active*. Finally, if *b* is a master block (all master blocks have dependents), then *b* is *quiescent* if *b* is complete, and all the direct dependents of *b* are quiescent.

Note that if a task *t* is quiescent and waiting on a terminate alternative, it *is* not valid to terminate it, since, for example, its master may not be complete and may still call *t*. We can, however, make the following claim.

Proposition 1. If a master *m* is quiescent and *m* is complete (i.e., not waiting at a terminate alternative), then *m* may be terminated .

Proof - Because of the recursive definition of quiescence, any task that depends on *m* either directly or indirectly is either complete or waiting at a terminate alternative. Thus *m* may be terminated according to ARM 9.4. □

The following proposition allows us to implement termination in a simple distributed fashion.

Proposition 2. While a master *m* is quiescent, no dependents of *m* can be called, and no new dependents of *m* can be created[13].

Therefore, the only way for *m* to become active (assuming *m* is waiting on a terminate alternative) is for another task to invoke *m* directly. The implication of Proposition 2 is that once a termination wave has begun, no task from outside the wave may call a task being terminated in the wave. Thus it is not necessary for the tasking system to lock out other tasks during a termination wave, i.e., for a termination wave to be implemented as an indivisible operation. Furthermore, the bookkeeping needed to check for and initiate termination can be performed in a relatively local manner, as described in §4.4.6 and §4.4.7. We first give a proof of Proposition 2.

Proof - We will prove the proposition in two parts: first the absence of callers, and then the absence of new dependents.

No Direct Dependents can be Called

Let *t* be a dependent of a quiescent master m, and suppose there is another task *t1* that invokes *t*. Then *t1* cannot be a dependent of *m*, since all dependents of *m* are quiescent. First, assume that *t* is a task object. For *t1* to call t, it must be the case that *m*, or a dependent of *m* that can access *t*, passes *t* to *t1* through a sequence of one or more entry calls to a task that is not in the dependency tree rooted at *m*. (If *m* is the direct master, *t* is declared in *m*, and is not otherwise accessible by name outside the dependency tree of *m*.) If *m* is not the direct master, *t* is declared in an inner block or dependent task of *m*. All dependents of the direct master of *t* are also dependents of *m*, and so again *t* is not accessible by name outside the dependency tree of *m*.) Let *o* be the task that depends on *m* and passes *t* to an outer dependency tree. Task objects are of limited type; therefore, *t* may not be passed by assignment (e.g., to a global variable). For *t1* to be able to reference *t* (passed as an in parameter), it must therefore be the case that o is currently executing an entry call, and so cannot be quiescent (See figure 4.4 5). This is a contradiction □

No Direct Dependents can be Created

Suppose that a task *t1* creates a task *t*, which becomes a dependent of a quiescent master m

13 A select statement may contain a terminate alternative and an accept alternative for an interrupt entry Such an accept may be called by the device associated with the entry while the master of the task which contains the select is quiescent However, an implementation is permitted to impose further requirements for the selection of the terminate alternative of a select with an accept alternative for an interrupt entry Thus, we can consider devices associated with interrupt entries as active dependents (requiring that tasks which contain interrupt entries be explicitly aborted)

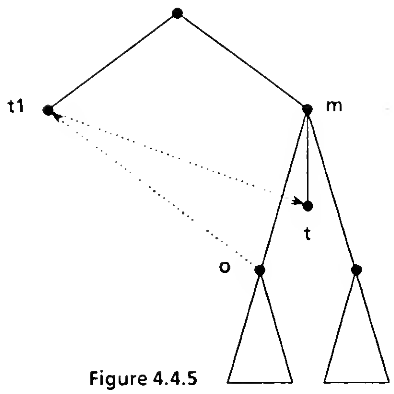


Figure 4.4.5

As above, the corresponding access type is declared in the direct master of *t*, and the access type definition must be visible to *t1*. Thus, *t1* must be a dependent of the direct master of *t*, and hence be a dependent of *m*. By hypothesis, such a task must be quiescent, and therefore cannot be executing an allocator. Thus while *m* is quiescent, no new dependent can be created. This completes the proof of Proposition 2. □

There are two allowable tasking idioms that we did not consider in the proof of proposition 2, namely, functions that return a task type, and tasks that are dependent on library packages. As pointed out by Rosen [Rosen 1983], tasks that are returned by functions must already be terminated, and therefore, do not invalidate the proof of proposition 2. We consider tasks that depend on library packages in the next section.

4.4.3 Library Packages

Another way for two tasks to gain visibility to a task pointer definition is for the task pointer type to be declared in a library package that they both name in *with* clauses. The rules of Ada specify that any library units named within a main program or its subunits are elaborated before the execution of the main program. The rules for task dependencies state that a task created by the evaluation of an allocator directly depends on the master that elaborated the corresponding access type definition. Thus, any tasks created by the evaluation of an allocator whose type definition appears in a library package directly depend on the library package.

The trouble with library packages is that their direct dependents can be passed to the direct dependents of the main program. Furthermore, new direct dependents of the library package can be created by the direct dependents of the main program. It would seem that the propositions are not true of tasks that are dependent on library packages.

However, the ARM 9.4 (13) states, *Consequently, termination of the main program awaits termination of any dependent task even if the corresponding task type is declared in a library package. On the other hand, termination of the main program does not await termination of tasks that depend on library packages; the language does not define whether such tasks are required to terminate.* So the main program need not be concerned with the termination of library packages and their dependents.

In order to have a uniform handling of task termination, SMARTS has an outermost *environment* task, on which the main program and library packages are dependent. The environment task waits for the tasks that are dependent on the library packages to terminate

4.4.4 Data structures for Termination

The data structures used to implement distributed block and task termination are given in this section. The actual termination algorithm is discussed in §4.4.6 and §4.4.7

The elements of the *TCB* used specifically for task termination are the two pointers, *first_deps* and *last_deps*, the two pointers, *master_task* and *master_block*, and the two counters, *no_term* and *num_deps*. The *first_deps* and *last_deps* are used to manage the dependent tasks list of a task. The *master_task* and *master_block* of a task point to its master task and master block (respectively). *no_term* counts the active direct dependents of a master task or a master block; *num_deps* counts the non-terminated direct dependents of a master block. *no_term* and *num_deps* are described in more detail below. The *status* field of the *TCB* is inspected to determine when a task is completed or terminated.

As tasks are terminated and destroyed by blocks, the *block_ptr* of a task is used during task termination. To inhibit aborts, during a termination wave the *rdv* flag of tasks that are waiting on select statements is reset to false. Finally, the *num_event* counter and *event* field are involved in task termination as tasks must sometimes block and unblock other tasks as they negotiate to terminate.

4.4.4.1 No_Term and Num_Deps Counters

To implement distributed task termination, we associate a *no_term* count with each master to keep track of the number of its direct (and inner) dependents that are not quiescent. (Similar counters have been used in uniprocessor implementations, e.g., [Riccardi and Baker 1984].) It is clear from the definition of quiescence, that if all direct (and inner) tasks are quiescent, then all dependents are also quiescent. The *no_term* count of a master task is defined as:

```
t.no_term  =  if t is complete, or waiting on a terminate alternative then
                      the number of active direct dependents of t
                  else
                      the number of active direct dependents of t + 1.
```

Thus *no_term* for a task includes the task itself in the count of active dependents. The extra 1 in *no_term* when a master is not complete or waiting on a terminate alternative allows us to implement the test for quiescence as an indivisible operation. Inner dependents are also counted, to handle the case illustrated by Example 4, Section 2. For a master block the *no_term* count is defined as:

m.no__term = if *m* is complete then

the number of active direct dependents of *m*

else

the number of active direct dependents of $m + 1$.

It follows from Proposition 1, for a master m, that when m.no__term = 0, m may be terminated.

Counters are also used to determine when it is safe to release the *TCB*'s of the dependents of a master block. A master block can only deallocate the *TCB*'s of its direct dependents after the direct dependents have deallocated the *TCB*'s of their direct dependents, etc. We associate a *num__deps* count with each master block to keep track of the number of its direct dependents that have not terminated. The *num__deps* count of a master block is defined as:

b.num__deps = if *b* is terminated, then

the number of non-terminated direct dependents of *b*

else

the number of non-terminated direct dependents of $b + 1$.

Again, the extra 1 in the *num__deps* counts when a master block is not terminated allows us to implement the test for the deallocation of dependent task as an indivisible operation

4.4.5 Termination of Master Blocks

In this section we deal only with the termination of master blocks and subprograms, that is, blocks and subprograms that have dependent tasks. Because we are only dealing with master blocks, the description of some of the actions performed when tasks are activated, become quiescent, etc., is omitted. Termination of tasks is somewhat more complicated, and is described in more detail in the next section. The *no__term* counter in the activation record of the master block indicates when the block can be exited: a brief summary of the actions performed on *no__term* is first specified, and code to implement this specification follows.

When a block b is entered its *no__term* and *num__deps* are set to 1. When a task t that is a direct dependent of a master block *b* is created, that parent of *t* enqueues *t* onto the dependent tasks list of *b*. Prior to activating *t*, the parent increments the *no__term* and *num__deps* counts of *m*. Similarly, when a task directly dependent on *b* accepts an entry call while waiting on a terminate alternative, and the task was quiescent before the call, it increments *b.no__term*.

When a block b is complete, *b.no__term* is decremented. If *b.no__term* is 0, then *b* is quiescent and can be terminated, otherwise the task executing *b* blocks. When a dependent task *t* becomes quiescent (either by completing or waiting on a terminate alternative, and decrementing its own counter to 0), it decrements *b.no__term*. If *t* discovers that its master block *b* is now quiescent, i.e., *b.no__term* is now 0, it notifies its master task (which is of course executing *b*) to resume and begin a termination wave for *b*.

Thus a master block *b* can terminate when *b* is complete and *b.no__term* becomes 0. Before terminating, *b* tells all its direct dependents that have not already terminated (or are in the processes of terminating) to terminate Each direct dependent does the same. (Note that when array of tasks are to be notified to terminate, their *SMARTS__node* is placed on a ready queue. Thus, the tasks will be terminated in parallel.) The dependents that must be notified to terminate will be waiting on selects with a terminate alternative. To ensure that the dependent is not aborted, the master attempts to disable the *rdv* flag of the dependent. Only if the master is successful will the master unblock the dependent. (Otherwise, the dependent will be unblocked by its necessarily successful aborter.)

When a master block *b* terminates, it decrements *b.num_deps*. If *num_deps* is 0, then *b* can deallocate the *TCB*'s of its direct dependents. After a task *t* that is a dependent of *b* deallocates the *TCB*'s of its direct dependents (i.e., the tasks on its dependent tasks list), *t* decrements *b.num_deps*. If *t* discovers that it is the last dependent of the master block *b* to deallocate the *TCB*'s of its dependents, i.e., *b.num_deps* is now 0, *t* notifies its master task to resume and begin deallocating the *TCB*'s of the direct dependents of *b*.

Since it is the master task that is unblocked after a dependent discovers that the *num_deps* of its master block is 0, it is imperative that the master block itself be included in the *num_deps*. If the *status* of the master task was used to indicate that a master block had terminated, i.e., set to *terminate_block*, then a dependent of a different master block (but same master task) could mistakenly assume that block being terminated was its master block, and prematurely unblock the master task.

4.4.6 Termination of Master Tasks

The operations involved in task termination are similar to those for block and subprogram termination There are, however, more complications. Once a block or subprogram becomes quiescent it cannot become active again. In contrast, a quiescent task that is waiting on a terminate alternative may become active by being called by a master, sibling, or sibling dependent Also, unlike a block, a task has masters that must be updated when the status of a task changes The *no_term* counter of a task *t* is incremented as follows

(i) When t is activated, the activator of *t* sets *t no_ term* to 1

(ii) If *t* is waiting on a terminate alternative and enters a rendezvous, it increments *no_term*.

(iii) Finally, *t.no_term* is incremented by a direct or inner dependent when the dependent is activated or when the dependent changes from quiescent to active. (See §4.3 for details on task activation.)

A significant observation is that when a quiescent task becomes active (or a task is created), it is only necessary to propagate the information one level up a dependency tree – i.e., to its master task (and/or master block). This follows from Proposition 2: if a task *t* is invoked, it must be the case that its master task is not quiescent, and hence no change of state (from quiescent to active) can occur in the master task. Consequently, no further updating nor checking need be performed. A similar remark applies when a task is dynamically created.

No_term counters are incremented as follows. When a task *t* completes or waits on a terminate alternative, it sets its *status* (to *complete* or *select_term*, respectively) and decrements its *no_term*. If in so doing it becomes quiescent, it also decrements *no_term* of its master task. When a task *t* discovers that its master task *m* is now quiescent and *m* is complete, then *t* notifies m to begin a termination wave; if *m* is waiting on a terminate alternative, then recursively *t* decrements masters of its master task until either of the following occurs:

- *t* finds a master task *m* that is not quiescent, i.e., $m.no\_term > 0$, (in which case *t* either blocks if it is not complete or terminates its own dependents if it is complete),

- *t* finds a master task *m* that is quiescent, i.e., $no\_term = 0$, and complete, i.e., $m.status = complete$, and can therefore start a downward termination wave.

A race condition is possible here because the check for quiescence and completion is not an indivisible operation. After the first test for quiescence but before *m.status* is checked, *m* may be invoked and a dependent *t1* of *m* activated. If *m* subsequently completes, the second test (for completeness) will succeed, even though *m* is no longer quiescent. Therefore, an additional test for quiescence, i.e., that $m.no\_term = 0$, is necessary.

It is possible that more than one task finds that a master task is quiescent and complete[14]. To ensure that the termination wave is initiated only once the *status* field of the *TCB* of the master task is used. After a task discovers that a master task is quiescent and complete it sets the *status* of the master task to *terminated* with a fetch&store. (This is the only time a task sets the status of another task.). If *status* was *completed*, then the task initiates the termination wave. When a master task discovers that its *no_term* is 0, then the master task also sets its *status* to *terminated* with a fetch&store; if the *status* had been reset by a dependent, then the master clears the pending unblock (the master task is already executing).

The recursive decrementing of *no_term* and checking for quiescence is the only time that information must be propagated for more than one level in a dependency tree. It need not, however, be an indivisible operation, nor be performed in a critical section. Furthermore, propagation and checking need only be performed up a single branch in the dependency tree. No subsequent rechecking down the tree is required. These observations follow from Proposition 2: while a task *t* is updating an ancestor *m* and *m* is quiescent, the status of all the dependents of *m* cannot change.

However, a termination wave is a complex affair: the task that starts the wave (the master task awakened by step 2 above) must both recursively update its masters' *no_terms* since it has become quiescent, and notify its dependents to terminate.

[14] To see how this can occur, consider Example 3 in §4.4.2. Suppose task *t2* completes, finds its master *dependent* waiting to terminate, and then decrements the counter of *master*. Suppose *master* is waiting on a terminate alternative, and that its counter becomes 0. Then, another task awakens *master*, which then calls entry *e* of *dependent*, and *dependent* creates a new task *t3*. Next *master* completes, *t3* completes, finds *dependent* quiescent, and updates the counter of *master*. If *t2* has done nothing in the meantime, both *t2* and *t3* will find *master* complete and quiescent, and hence, signal *master* to start a termination wave. In a similar scenario, it is also possible that both a master task and a dependent find that the master is both quiescent and complete.

After a task that was waiting on a terminate alternative participates in a termination wave, the task must wait for its dependents to deallocate the *TCB*'s of their dependent by block. By associating a *num__deps* counter with master tasks (as well as master blocks) we could eliminate this serialization as follows: Prior to activating a dependent of a master task, the *num__deps* count of the master task is incremented. After a dependentof a master task deallocates the *TCB*'s of its dependents, the *num__deps* count of the master task of a dependent is decremented. The test for unblocking a master task becomes (i.e., the procedure *check__done*): the *num__deps* of the master block or the master task is 0 (after being decremented). A task that is waiting on a select with a terminate alternative in a nested block will not have completed enclosing blocks, and hence, will not have decremented the *num__deps* counts of any master blocks. Thus, the master task will only be unblocked when all of its dependents have deallocated the *TCB*'s of their dependents. The overhead of this scheme is two fetch&add's per task, which seems excessive given that selects with terminate alternatives are not expected to be deeply nested.

The code for terminating blocks and task is given below.

4.4.7 Code for Block and Task Termination

-- Code for Completion of Tasks and Blocks

```
procedure complete_block(TASK; BLOCK) is
-- Last instruction executed by a block
begin
    if fetch_and_add(BLOCK.NO_TERM, -1) /= 1 then
    -- We have active dependents, so wait for them to be quiescent.
        block_task(TASK);
    end if;
    term_wave(BLOCK.FIRST_DEPS); -- Start a termination wave of our dependents.
    if fetch_and_add(BLOCK.NUM_DEPS, -1) /= 1 then
    -- Wait for our dependents to free the TCBs of their dependents.
        block_task(TASK);
    end if;
    -- Release the tcbs of my direct dependents.
    uncreate_tasks(BLOCK.FIRST_DEP);
end complete_block;
```

```
procedure complete(TASK) is
-- The task has completed its execution.
begin
    write(TASK.STATUS, COMPLETE);
    --Raise TASKING_ERROR in all the tasks waiting on entry queues of TASK
    purge_rdv(TASK);
    if fetch_and_add(TASK.NO_TERM, -1) /= 1
    or else fetch&store(TASK.STATUS, TERMINATED) = TERMINATED then
    -- Wait for our kids to to be quiescent.
        block_task(TASK);
    end if;
    terminate(TASK); -- Clean up.
end complete;
```

-- Code for Termination of Tasks.

```
procedure terminate(TASK) is
-- Last procedure executed by a task.  All of my dependents have been notified to
terminate.
begin
    -- BLOCK_PTR is the outermost scope of the task.
    if fetch_and_add(TASK.BLOCK_PTR.NUM_DEPS, -1) /= 1 then
    -- Start a termination wave and then wait for my dependents to be done.
        term_wave(TASK.BLOCK_PTR.FIRST_DEPS);
        block_task(TASK);
    end if;
    -- Release the tcbs of my direct dependents.
    uncreate_tasks(BLOCK_PTR.FIRST_DEP);

    -- Note that, if the task has dependents then the task may have been unblocked,
    -- and not have executed the corresponding unblock.  This is fine, however,
    -- because the task never blocks again.
    write(TASK.STATUS, TERMINATED);
    free(TASK.STACK_PTR) -- Free my stack space.
    -- Check if our master task, or master block is completed.
    check_done(TASK.MASTER_TASK, TASK.MASTER_BLOCK);
    initialize_pe; -- Get a new task.
end terminate;

procedure term_wave(TASK_LIST) is
-- Send terminate messages to the (necessarily quiescent) dependents of a block.
-- Note: This could be put on the ready queue and done in parallel.
begin
    while TASK_LIST /= null loop
        case read(TASK_LIST.STATUS) is
        when ACTIVATE =>
        -- If a multi-item (last action was activation).
            write(TASK_LIST.STATUS, TERMINATE);
            write(TASK_LIST.MULT, DEPENDENT.TEMPLATE_PTR.MULT).
            fetch_and_add(NUM_RUNNABLE, TASK_LIST.MULT). -- Schedule will adjust
            ready_enqueue(TASK_LIST, TASK_LIST.TCB_PTRS(1).PRIORITY).
        when SELECT_TERM =>
```

```
            if fetch_and_store(TASK_LIST.RDV, false) = true then
            -- Ensure that task is only unblocked once, by term_wave or abort.
               unblock_task(TASK_LIST, TERMINATE);
            end if;
         others =>
            null; -- Already completed or terminated.
         end case;
         TASK_LIST := read(TASK_LIST.SIBLING);
      end loop;
   end term_wave;

   procedure terminateme(TASK) is
   -- A task that was waiting on a select with a terminate alternative, has been
   -- told to terminate.
   begin
      BLOCK := read(TASK.BLOCK_PTR);
      -- Send termination messages to dependents by block.
      while BLOCK /= null loop
      -- Notify all the dependents of block to terminate.
         term_wave(BLOCK.FIRST_DEP);
         BLOCK := read(BLOCK.BLOCK_PTR);
      end loop;
      BLOCK := read(TASK.BLOCK_PTR);
      -- Release space of my dependents by block.
      while read(BLOCK) /= null loop
         if fetch_and_add(TASK.BLOCK_PTR.NUM_DEPS, -1) /= 1 then
         -- Wait for the dependents of the block to free the TCB's of their dependents.
            block_task(TASK);
         end if;
         -- Release the tcbs of the block's direct dependents.
         uncreate_tasks(BLOCK.FIRST_DEP);
         BLOCK := read(BLOCK.BLOCK_PTR);
      end loop;

      -- Terminate the task.
      write(TASK.STATUS, TERMINATED);
      free(TASK.STACK_PTR) -- Free my stack space.
      -- Check if our master task, or master block is completed.
      check_done(TASK.MASTER_TASK, TASK.MASTER_BLOCK);
      initialize_pe; -- Get a new task.
   end terminateme;
```

-- Utility Routines for Completion and Termination of Blocks and Tasks.

```
   procedure check_term(TASK; BLOCK) is
   -- Called by a dependent of TASK (and BLOCK) whose NO_TERM is 0.
   begin
      if fetch_and_add(TASK.NO_TERM, -1) = 1 then
         if read(TASK.STATUS) = COMPLETE then
            if read(TASK.NO_TERM) = 0
            and then fetch_and_add(BLOCK.NO_TERM, -1) = 1 then
            -- Recheck NO_TERM to avoid race condition, and then make certain
            -- we are the first
```

```
                if fetch_and_store(TASK.STATUS, TERMINATED) = COMPLETE then
                   unblock_task(TASK, TERMINATE);
                end if;
             end if;
          else
          -- TASK is waiting on a terminate alternative  so check TASK's master.
             check_term(TASK.MASTER_TASK, TASK.MASTER_BLOCK) ;
          end if ;
       else
       -- TASK is not quiescent,  but BLOCK may be complete.
          if fetch&add(BLOCK.NO_TERM, -1) = 0 then
          -- the last dependent and BLOCK complete.
             unblock_task(TASK, TERMINATE);  -- TASK is executing BLOCK.
          end if;
       end if;
    end check_term;

    procedure check_unterm(TASK) is
    --TASK was waiting on a terminate alternative, but has been called
    begin
       if fetch_and_add(TASK.NO_TERM, 1) = 0 then
       -- TASK was quiescent, so update its master
          add(TASK.MASTER_TASK, 1);
          if read(TASK.MASTER_TASK) /= read(TASK.MASTER_BLOCK) then
             fetch_and_add(TASK.MASTER_BLOCK.NO_TERM, 1);
          end if;
       end if;
    end check_unterm;

    procedure purge_rdv(TASK) is
    -- Raise tasking error in all the tasks waiting on entry queues of TASK.
    begin
       for I in 1..read(TASK.NUM_ENTRIES) loop
       -- Note: no new tasks will be added since my abnormal flag is set.
          owner_lock(TASK.ENTRY[I]); -- No task can be enqueued to this queue.
          CALLER := read(TASK.ENTRY[I].FIRST);
          while CALLER /= null loop
             write(CALLER.EXCEPTION, TASKING_ERROR);
             unblock_task(CALLER, END_RDV);
             CALLER := CALLER.NEXT;
          end loop.
          owner_unlock(TASK.ENTRY[I]);
       end loop.
    end purge_rdv;

    procedure check_done(TASK, BLOCK) is
    -- Check if we are the last dependent of BLOCK to release space for our dependents
    begin
       if fetch_and_add(BLOCK.NUM_DEPS, -1) = 1 then
       -- Our master block is done
          unblock_task(TASK, DONE).
       end if.
    end check_done.
```

```
procedure uncreate_tasks(TASK_LIST) is
-- Free the TCB's of the task list.
begin
    -- Terminate the tasks
    while TASK_LIST /= null loop
        NEXT := read(TASK_LIST.SIBLING);
        free(TASK_LIST);
        TASK_LIST := NEXT;
    end loop;
end uncreate_tasks;
```

-- Code for TERMINATED.

```
procedure is_terminated(TASK) return boolean is
begin
    return (read(TASK.STATUS) = TERMINATED);
end is_terminated;
```

4.4.8 Absence of Race Conditions in Task and Block Termination

We give a proof that the *no_term* counters correctly implement distributed termination of master tasks and blocks. The proof that the *num_deps* counters safely implement the deallocation of the *TCB*'s of task is similar, and, as it is much simpler, is omitted. The potential for race conditions exist because more than one dependent task may access a task control block concurrently, and some of the actions involved in bookkeeping are not indivisible.

To show the absence of race conditions, we must rule out three types of anomalous behavior: premature termination wave (before its preconditions are satisfied), failure to initiate a termination wave (when its preconditions are satisfied), and more than one initiation of a termination wave (for a single instance of its preconditions). An implementation that possesses the first two of these properties is said to exhibit *safety* and *liveness* (respectively). We will call the third condition *at-most-once semantics*.

Safety and liveness have been identified as essential properties for events such as task termination detection [Apt 1986]. At-most-once is one of several semantics used to classify client's requests of servers in distributed operating systems where, due to equipment failures, messages can sometimes be lost [Tanenbaum 1987]. Given an unreliable environment, when a client does not receive a reply after requesting a service from a server, the client does not know if the service has been performed, i.e., whether the request or the reply was lost. Hence, the client must request the service of the server again (even though the service may have already been performed). At-most-once semantics guarantees that the service will not be performed more than once even when the client sends more than one request.

While we do not take into account hardware or operating system failures (i.e., we are not addressing reliability), because our RTS is distributed we must consider the possibility that more than one SMARTS processor may discover that the preconditions for termination are satisfied and attempt to initiate a termination wave.

4.4.8.1 Safety

A potential race condition involves the checks that the actual termination conditions for initiating a termination wave, completeness and quiescence, are met. These are two separate checks, and hence not an indivisible operation. These checks are performed when a dependent decrements the counter of a master task (in procedure *check_term*), and when a master task completes and checks its own counter. In the first case, the dependent checks for *no_term* = 0 twice. Therefore, if the state changes between the check of *no_term* and the check for completeness, the dependent will not initiate a termination wave prematurely. (Note that once a task is complete its status cannot change.) In the latter case, the master task sets its status to complete before decrementing its *no_term* counter. Thus, either the master or the dependent will discover correctly the case in which a termination wave can begin

Given that either a master task *m* or its dependents correctly discover when m is complete and *m.no_term* is 0, we must prove that when these conditions are met a termination wave of the dependents of *m* can be initiated. The following proposition is useful in showing that a termination wave will not be started prematurely.

Proposition 3. While a task *t* is not waiting on a terminate alternative nor completed, *no_term* of its master cannot become 0.

Proof. This follows directly from the contrapositive of Proposition 2: While dependents of a master m can be called or new dependents of m can be created, *m* cannot be quiescent. □

A termination wave of the dependents of *m* will be initiated after 0 or more calls to *check_term* are made by the task that initiates the termination wave. We will prove that no termination waves are started prematurely by induction on the number of these calls to *check_term*.

Basis. When *i* is 0, then it is *m* that discovers that a termination wave should be initiated. From Proposition 1 we can deduce that a termination wave can be initiated.

Induction Hypothesis. When it is discovered that a master task *m* is quiescent, i.e., m.status is complete and *m.no_term* = 0, via *i* calls to *check_term* then a termination wave of the tasks in the dependency tree rooted at m can be initiated.

Step. Suppose the termination wave is discovered by a task *t* via $i+1$ recursive calls of *check_term*. While *t* is active we can deduce from Proposition 3 that *m.no_term* is not 0, and hence, no termination wave will be initiated before *t* calls *check_term*. The first call made by *t* to *check_term* must find *t.master_task* quiescent, since we know *t* eventually initiates a termination wave. From Proposition 2 we can deduce that while *t.master* is quiescent no dependents of *t.master* will be called or created, and hence, disturb the conditions for initiating a termination wave. We complete the proof by appealing to the Induction Hypothesis for next i calls. □

4.4.8.2 Liveness

Every time *no_term* of a task is decremented it is tested for being 0. When a dependent task that is about to wait on a terminate alternative finds its *no_term* = 0, it calls *check_term* to test its masters. In *check_term* a dependent will check twice if the *no_term* of a master task is 0; before and after it checks if the status of the master task is complete. A master always sets its status prior to decrementing or incrementing its *no_term* count. In particular a master task sets its status to complete before decrementing its *no_term* counter. Thus, either the dependent or the master task will detect when a termination wave can be initiated.□

4.4.8.3 At-Most-Once Semantics

A termination wave is discovered by either a master task *m* or one of its dependents. At most one dependent will succeed in discovering that the conditions for initiating a termination wave and executing fetch&store(*m.status*, *terminated*) = *completed*. When *m* completes and finds *m.no_term* is 0, it also executing fetch&store(*m.status*, *terminated*) = *completed*; if *m.status* was already set to *terminated* then a dependent has also discovered that *m* is quiescent, so *m* clears the pending unblock. Otherwise *m* has succeeded in inhibiting a dependent from executing the *unblock_task*(*m*, *terminate*). Therefore, exactly one dependent or *m* itself will initiate a termination wave of the tasks in the dependency tree rooted at *m*. □

4.5 Time Management

The execution of a delay statement by a task suspends the execution of the task for at least the duration of the delay. For example, the execution of the statement:

delay 4.5;

by a task will cause the execution of the task to be suspended for at least 4.5 seconds.

Delays may also be used as the alternatives of select statements - both selective waits (select statements whose alternatives consist of accept statements and delay statements) and timed entry calls (select statements whose two alternatives are an entry call and a delay statement); if a rendezvous is begun before the delay expires, the delay is canceled. A select statement with a terminate alternative, however, cannot have a delay alternative.

When a task whose execution is suspended at a delay statement is aborted, the delay is canceled and the task becomes completed.

Delays specify a relative time, while the Ada function *clock* deals with absolute time: clock returns the current time of day. The value returned by clock is of the predefined type *time*. Although both *clock* and *time* deal with time and are included in the library package *calendar*, their relationship is tenuous. For instance, delay and clock may have different precisions [AI-00201]. So after a task executes the sequence:

```
NOW := clock + 0.1;
delay 0.1;
TEST := (NOW <= clock);
```

test is not required to be **true** since clock might be updated less than 10 times a second.

Ada does not provide a mechanism for delaying a task until an absolute time. Such an ability is useful for implementing tasks that handle periodic events. The calendar package supported by SMARTS, therefore, includes the function *alarm(when)* for suspending a task until the time *when*. *when* must be of the predefined type *time*. A task that executes an *alarm*(*when*) will be suspended until after the value returned by *clock* is greater than *when* The body of *alarm* can be written in Ada as,

```
    DURATION := WHEN - clock;
    delay DURATION;
```

However, since this code could be interrupted after the first statement (causing the task to be suspended longer than necessary) SMARTS treats the function as though it was a single statement (i.e., atomically).

In the next section we describe the implementation of delay statements. We also discuss the way in which aborts and rendezvous disable delays. The implementation of rendezvous and aborts are discussed in detail in §4.6 and §4.7.

To implement delays, we make use three kernel functions: *start__timer*, *stop__timer* and *read__time*. *start__timer*(*value*) causes a (hardware) timer interrupt to be issued in *value* seconds; *stop__timer*(*value*) cancels a pending timer interrupt (if there is one) and assigns to *value* the time that was remaining until the interrupt. *time* returns the time of day. Whereas *start__timer* and *stop__timer* take relative times as their parameters and communicate with a processor's local clock, *read__time* returns an absolute time and communicates with the global clock. These functions are provided by most operating systems, for example, in UNIX they correspond to the *setitimer*(*ITIMER__REAL*, *value*), *getitimer*(*ITIMER__REAL*, *value*, *ovalue*) and *gettimeofday*(*time*, *time__zone*) system calls

We require that processors read a global clock for *read__time* to avoid problems when the local clocks of the processors get (temporarily) out of synch with each other If each processor relied on its local clock for absolute times, then a task that was context switched between two consecutive calls to clock might get an earlier value returned on the second call than on the first by virtue of having read the clock of a different processor. If having all the processors of a highly parallel machine access a global clock is not feasible, then SMARTS can refuse to support the clock function under AI 00325 (which deals with implementation dependent limitations)

We have chosen to implement the interrupt handlers for the local clocks of the processors as interrupt entries (see §4.6). By associating the address of the clock with the entry *time__out* of the task *timer__handler*, timer interrupts issued by the clock will act as entry calls to *timer__handler.time__out*. Each SMARTS processor has its own copy of the task *timer__handler* with *time__out* associated with the address of its local clock. An alternative implementation of the timer interrupt handlers would have been to use the kernel function *install*. (Indeed, SMARTS's implementation of interrupt entries employs *install*.) *install(device, handler)* installs *handler* as the interrupt handler for *device*, i.e., as the routine to be executed (without software intervention) when *device* issues an interrupt. Most operating systems provide a system call for specifying interrupt handlers. In UNIX a timer interrupt handler can be specified using the system call *signal(SIGALARM, time__handler)*, where *time__handler* is a subprogram containing the code to be executed after a timer interrupt is generated.

4.5.1 Data Structures for Time Management

The *time__node* of a task is the primary data structure used for time management. The *time__node__ptr* of the *TCB* of a task points to the current *time__node* of a task. A *time__node* contains four fields: *when, tag, next* and *task__ptr*. *when* is set to the absolute time at which the current delay is set to expire; the *tag* is used to disable the delay when the task is aborted or a rendezvous is begun before the delay has expired; the *next* pointer points to the next *time__node* on the same time chain; the *task__ptr* points to the task that owns the *time__node*.

Each SMARTS processor that executes non-preemptively on a (physical) processor (see §4.2) maintains a time chain The chain is a linked list of *time__nodes* implemented with fetch&store's (see §4.3.1); this implementation was chosen since it allows for concurrent interior removals (a time__node must be removed if the task that owns it is aborted or enters

Element	Use
when	The absolute time to be awakened.
tag	Used to cancel the delay Initially set to status of the task.
next	The next time_node on the time chain.
task_ptr	The owner of the time_item.

Table 4.5.1: Time_node Data Structure

a rendezvous); the *time__node*'s on a time chain are kept in monotonically increasing order according to their *when* fields. The *clock__head* pointer of a SMARTS processor points to the first *time__node* on its time chain.

The other data structures belonging to a SMARTS processor that are used in time management are *next__slice*, *preempt__flag* and *slice*. They are used to implement pre-emptive scheduling: *next__slice* is the time at which the SMARTS processor should next be pre-empted; *slice* is the time between pre-emptions; and *preempt__flag* is used to software disable pre-emption.

Since the execution of a delay statement is an abort synchronization point, i.e., abnormal tasks suspended at delay statements must become completed, the *abnormal* flag of a task is also used in the implementation of delays.

4.5.2 Execution of a Delays and Alarms

When a task executes a delay statement it first inspects its *abnormal* flag If that task has not been aborted, then the task enqueues its *time__node* onto the time chain of the SMARTS processor that is executing the task This enables a time out set to expire after the duration of the delay seconds. (The actual enabling of time outs is described in the next subsection) The task then sets its *status* to the type of delay statement that the task is executing which will be either *wait*, *timed__call* or *timed__select* The task can now be unblocked by an aborter Thus, the task must reinspect its *abnormal* flag to ensure that it does not miss an abort (see

§4.7.1.1). If the *abnormal* flag is set, then the task attempts to cancel the delay. Otherwise, the task blocks.

The task will be unblocked if it either times out, is aborted, or enters a rendezvous; when a task that is waiting on a delay is unblocked because of an abort or a rendezvous its time out will have (necessarily) been disabled. In next three subsections we describe how time outs are enabled, timed out, and disabled.

When the task is unblocked it checks its *abnormal* flag to see if it was aborted during the delay; if the *abnormal* flag is not set (i.e., it was not aborted) then the task checks its *event* field to determine which event (time out or rendezvous) caused it to become unblocked.

The implementation of the library function alarm is almost identical to the implementation of delay. The only difference is that for alarm we do not need to calculate the absolute time at which the delay is to expire since it is already specified. (The absolute expiration time is needed because the *time_node*s an a time chain are kept monotonically increasing according to the time at which they are to expire.)

4.5.3 Enabling Time Outs

If the *time_node* of a task is currently acting as a place holder in a time chain when the task executes a delay statement, then a new *time_node* is created and assigned to the *time_node_ptr* of the task. The *when* field of the *time_node* is set to the actual time at which the delay will expire (the current time plus the duration). The *tag* field of the *time_node* is set to the type of delay statement that the task is executing which will be either *wait, timed_call* or *timed_select*. (The *status* of the task will be set to the same value after the *time_node* is enqueued.) The *time_node* of the task is then enqueued onto the time chain of the SMARTS processor on which the task is running. the time out is now enabled, so the task blocks

The time chain of a SMARTS processor is a singly linked list of the *time_nodes* (kept in ascending order by their *when* fields) of the tasks that have executed delay statements while being executed by the SMARTS processor The timer of the processor that is executing the SMARTS processor is set to expire (i e., *interrupt* the processor) in the number of seconds remaining for the delay of the first *time_node* on the time chain of the SMARTS processor (i.e., *time* – *when*). When the timer expires a timer interrupt is generated and the entry *time_out* of the task *timer_handler* (described in the next section) is called

While a *time_node* is being enqueued to the time chain of a SMARTS processor, timer interrupts are disabled for the processor that is executing the SMARTS processor After a *time_node* has been enqueued the timer is re-enabled and set for the number of seconds now remaining for the delay of the first *time_node* on the time chain. This will cause some slippage in time, since some time elapses between calculating the number of seconds left and setting the timer, however, this implementation still satisfies the ARM since delays are only required to suspend a task for *at least* as long as their duration To recover this lost time, when a timer interrupt occurs for the first *time_node* on a time chain all of the *time_nodes* on the time chain whose delays should have expired (i.e., whose *when* field are less than or equal to *time*) are timed out

The operations performed on time chains are dequeues from the front of the chain of *time_nodes* that have timed out or been effectively removed, enqueues of *time_nodes* to their appropriate position in the time chain, and interior removals of *time_nodes* In addition, a timer interrupt must be set up for the first *time_node* on the time chain We store in the *when* field of a *time_node* the actual time at which the delay is to expire An alternative implementation is to store the relative time at which the delay is to expire, i e , the amount of time after the previous *time_node* (in the chain) expires Setting up timer interrupts is slightly more complicated with the former scheme than with the latter scheme, while the actual enqueues and the dequeues of *time_nodes* are simpler with the latter scheme than with the former scheme The cost of an enqueue is proportional to the position in the time chain of the *time__node* being enqueued. Since a timer a interrupt and dequeue each occur at most once per *time__node* (when it makes its way to the front of the chain), whereas a *time__node* may have to be visited many times during the enqueue of other *time__nodes* the former scheme is preferable to the latter.

In SMARTS, the correct position for a *time__node* being added to the time chain is found using a simple linear search; it was not felt that time chains would be long enough to justify using a more complicated (but better time complexity) search method, although there is no reason why one could not be used. While the search is being performed *time__nodes* that have been effectively removed (disabled) and left in the time chain as place holders are actually removed.

4.5.4 Time Outs

The entry *time__out* of the task *timer__handler* is installed as the timer interrupt handler of a processor by associating the address of the timer of the processor with the entry. After being invoked because of a timer interrupt, *time__out* attempts to complete the time out of the first *time__node* on the time chain of the SMARTS processor. It begins by resetting the *tag* of the *time__node* with a fetch&store(*time__node.tag*, *time__out*). What is done next depends on the value returned by the fetch&store (Recall that when a time__node is added to a time chain its *tag* is set to the status of its owner, i.e., *wait*, *timed__call* or *timed__select*.).

If the value returned is *removed*, then the time out fails – the *tag* has been set to *removed* by the task that has caused the owner of the *time__node* to be aborted or to enter a rendezvous.

If the value returned is *wait*, then the time out is successful (the task was executing a simple delay statement).

If the value returned is *timed__call*, then the time out is successful if the owner of the *time__node* can also be removed from the entry queue that the owner is on.

If the value returned is *timed__select*, then the time out is successful if the *rdv* tag of the owner of the *time__node* can be reset, i.e., if we can disable the rendezvous.

After a timer interrupt has (successfully or unsuccessfully) reset the *rdv* tag of a task or removed a task from an entry queue it sets the *tag* of the *time__node* to *dequeued*, signaling that the dequeue operation is complete.

All the *time__node*'s at the front of the time chain whose delays have expired during this timer interval (either because of the time slippage discussed in §4.5.3 or because their delay was was equal to to that of the first *time__node* on the time chain) are also timed out.

If the time slice of the SMARTS processor is up, then the *timer__handler* will attempt to force a context switch, i.e., pre-empt the SMARTS processor. The *timer__handler* first increments the *preempt__flag* of the SMARTS processor. If the old value was 0, then interrupts are not software disabled and the *timer__interrupt* can force a context switch. If the old value was not 0, then either a context switch is in progress (and hence, need not be forced) or the current task is executing a ready enqueue operation. In the later case, the task will voluntarily relinquish the SMARTS processor when the enqueue operation is completed (after discovering that the *preempt__flag* has been incremented).

If a time slice has expired the next time slice is set up The next timer interrupt is then set up and enabled

4.5.5 Disabling Time Outs

The purpose of the *tag* field of a time__node is to allow *time__nodes* to be removed from the time chain of a SMARTS processor (and hence, to disable time outs) without interrupting the execution of the processor that is executing the SMARTS processor A *time__node* is removed from the interior of a time chain when after executing a delay its owner participates in a rendezvous or is aborted

Interior removals are implemented as described in §2.2.1. The node is not actually removed – just tagged as such – and left in the queue as a place holder. The *time__node* will not be placed on another time chain unless it has actually been dequeued, i.e., its *tag* set to *dequeued*. The owner of a *time__node* that is left in a time chain as a place holder, must allocate a new *time__node* if it executes another delay before its current *time__node* has been dequeued.

When a task *a* aborts a task *t* that is executing a simple delay statement, *a* tries to remove the *time__node* of *t* from whatever time chain it is on by resetting the *tag* of the *time__node* with a fetch&store(*t.time__node__ptr.tag*, *removed*). If the value returned by the fetch&store is *wait* (the status of *t*) then the *time__node* has been effectively removed, the time out for *t* has been disabled, and the *time__node* is left in the time chain as a place holder. If some value other than *wait* is returned by the fetch&store then the removal fails; the abort, however, does not fail as *t* will discover that its *abnormal* flag has been set when it is unblocked by the time out.

When a task *ar* aborts or wants to engage in a rendezvous with a task *t* that is executing a conditional wait, *ar* tries to remove *t* from the entry queue that it is on; if *ar* is successful in removing *t* then the time out for *t* has been disabled. When *t* is unblocked, *t* will attempt to remove its *time__node* from whatever time chain it is on, however, so that a timer interrupt will not be set for *t* unnecessarily.

When a task *ar* aborts or wants to engage in a rendezvous with a task *t* that is executing a selective wait, *ar* tries to reset the *rdv* tag of *t*; if *ar* is successful in resetting the *rdv* tag then the time out for *t* has been effectively disabled. When *t* is unblocked, *t* will attempt to remove its *time__node* from whatever time chain it is on, however, so that a timer interrupt will not be set for *t* unnecessarily.

In the next section we give the actual code for the execution of simple delay statements, enabling time outs, time outs and disabling time outs. The code for the execution of the delay alternatives of select statements is given in §4.6.

4.5.6 Code for Time Management

-- The Execution of Delay or Alarm Statements.

```
procedure wait(TASK; DELAY) is -- Execute a simple wait statement.
begin
    if read(TASK.ABNORMAL) then -- This is a synchronization point.
        abortme(TASK);
    end if;
    WHEN := DELAY + time; -- Change to absolute time.
    time_enqueue(TASK, WHEN, WAIT);
    write(TASK.STATUS, WAIT); -- Aborts are now possible.
    if read(TASK.ABNORMAL) and then time_remove(TASK.TIME_NODE_PTR) then
    -- I have been aborted and the time out is disabled (otherwise I'll be unblocked).
        write(TASK.TIME_NODE_PTR, null); -- Left in time to_enqueue.
        abortme(TASK);
    end if;
    -- Wait for delay to expire (or to be aborted).
    if timed_block_task(TASK, EVENT) then
        EVENT := fetch_and_store(TASK.EVENT, NULL_EVENT); -- Clear event.
    end if;
    write(TASK.STATUS, ACTIVE);
    if TASK.ABNORMAL then -- This is a synchronization point.
        if EVENT = ABORT then -- Left in time to_enqueue.
            write(TASK.TIME_NODE_PTR, null);
        end if;
        abortme(TASK);
    end if;
end wait;

procedure alarm(TASK; WHEN) is -- Execute the library subprogram alarm.
begin
    if read(TASK.ABNORMAL) then -- This is a synchronization point
        abortme(TASK);
    end if,
    if WHEN <= time then
        return;
    end if;
    time_enqueue(TASK, WHEN, WAIT);
    write(TASK.STATUS, WAIT); -- Aborts are now possible
    if read(TASK.ABNORMAL) and then time_remove(TASK.TIME_NODE_PTR) then
    -- I have been aborted and the time out is disabled (otherwise I'll be unblocked)
        write(TASK.TIME_NODE_PTR, null), -- Left in time to_enqueue
        abortme(TASK);
        end if;
        -- Wait for delay to expire (or to be aborted).
        timed_block_task(TASK, EVENT);
        write(TASK.STATUS, ACTIVE);
        if TASK.ABNORMAL then -- This is a synchronization point.
            if EVENT = ABORT then -- Left in time to_enqueue.
                write(TASK.TIME_NODE_PTR, null);
            end if;
            abortme(TASK);
        end if;
    end wait;
```

-- Code for Enabling Time Outs.

```
    procedure time_enqueue(TASK; WHEN; STATUS) is
    -- Add TASK to the time out chain of the SMARTS processor that is executing it.
    -- Called when a task executes a delay statement.
    begin
        -- Disable timer while we insert.
        -- VALUE is set to the time remaining on this interval.
        stop_timer(VALUE);
        NOW := read_timer;
        VALUE := VALUE + NOW; -- Minimize slippage.

        -- Set up the node to be inserted on the list.
        if read(TASK.TIME_NODE_PTR.TAG) /= DEQUEUED then
        -- Old node left in chain as a place holder.
            write(TASK.TIME_NODE_PTR, TIME_NODE_REC);
        end if;
        NEW_NODE := read(TASK.TIME_NODE_PTR);
        write(NEW_NODE.TAG, STATUS);
        write(NEW_NODE.WHEN, WHEN);

        -- Get rid of interior removals at the start of the list.
        while read(CLOCK_HEAD) /= and then read(CLOCK_HEAD.TAG) = REMOVED loop
            write(CLOCK_HEAD.TAG, DEQUEUED); -- Node can be reused.
            CLOCK_HEAD := read(CLOCK_HEAD.NEXT);
        end loop;

        -- Do a sequential search for NEW_NODE's position.
        if read(CLOCK_HEAD) = null or else read(WHEN) <= read(CLOCK_HEAD.WHEN) then
        -- Insert at the beginning of the list.
            write(NEW_NODE.NEXT, CLOCK_HEAD);
            CLOCK_HEAD := read(NEW_NODE);
        else
        -- Find node's place in list (and actually remove interior removals).
            BEFORE := read(CLOCK_HEAD);
            AFTER := read(BEFORE.NEXT);
            while AFTER /= null and then read(AFTER.WHEN) < NEW_NODE.WHEN loop
                if read(AFTER.NEXT) /= null and then read(AFTER.NEXT.TAG) = REMOVED then
                -- Interior removal.
                    write(AFTER.NEXT, AFTER.NEXT.NEXT);
                end if;
            write(AFTER, AFTER.NEXT);
        end loop;
        -- Insert NEW_NODE into the list
        write(NEW_NODE.NEXT, AFTER);
        write(BEFORE.NEXT, NEW_NODE);
    end if;

    -- The timer will expire in VALUE seconds (a negative VALUE times out
    -- immediately).
    VALUE := min(VALUE, read(CLOCK_HEAD.WHEN)) - read_time;
    start_timer(VALUE); -- Some slippage.
end time_enqueue;
```

-- Code for Time Outs.

```
task type timer_handler is
    entry time_out;
    for time_out use at TIMER_ADDRESS;
end timer_handler;

task body timer_handler is
begin
    select
        accept time_out  do  -- Catch timer interrupts.
        -- Time out all of the tasks waiting on this interval.

            while read(CLOCK_HEAD) /= null and then CLOCK_HEAD.WHEN <= read_time loop
            -- Try to claim the time_node.
                to_dequeue(CLOCK_HEAD);
                write(CLOCK_HEAD, CLOCK_HEAD.NEXT);
            end loop;

            -- Remove interior removals at the front of the chain..
            while read(CLOCK_HEAD) /= null
            and then read(CLOCK_HEAD.TAG) = REMOVED loop
                write(CLOCK_HEAD.TAG, DEQUEUED); -- Node can be reused.
                write(CLOCK_HEAD, CLOCK_HEAD.NEXT);
            end loop;

            -- SMARTS processor may have to be pre-empted.
            if read_time >= NEXT_SLICE then
                if fetch_and_add(PREEMPT_LOCK, 1) = 0 then
                -- If pre-emption not software disabled.
                    preempt_pe;
                    PREEMPT_LOCK := 0; -- Reset lock.
                end if;
                NEXT_SLICE := NEXT_SLICE + SLICE;
            end if.

            -- Set up next interrupt
            NEXT_WHEN := min(read(CLOCK_HEAD.WHEN), SLICE);
            VALUE := NEXT_WHEN - time; -- Some slippage.
            -- The timer will expire in VALUE seconds (a negative VALUE times out
                -- immediately).
                start_timer(VALUE);

            end time_out;
        else
            terminate; -- Terminate when the SMARTS processor completes
        end select;
    end timer_handler;

    procedure time_dequeue(NODE)  is
    --- Try to claim the first time_node on a time chain.
    begin
        case fetch_and_store(NODE.TAG, TIME_OUT) of
        -- Flag was originally set to the task's status.
        when REMOVED => -- It's already gone.
            write(NODE_PTR.TAG, DEQUEUED); -- The time out attempt is over.
        when TIMED_CALL => -- Try to remove from entry queue.
            if entry_remove(NODE.TASK) then
                write(NODE_PTR.TAG, DEQUEUED); -- The time out attempt is over.
                timed_unblock_task(NODE_PTR.TASK_PTR, TIME_OUT);
            else
                write(NODE_PTR.TAG, DEQUEUED); -- The time out attempt is over.
            end if;
        when TIMED_SELECT => -- Try to disable rendezvous.
            if fetch_and_store(NODE_PTR.TASK.RDV, false) = true then
                fetch_and_store(NODE_PTR.TAG, DEQUEUED); -- The time out attempt is over.
                timed_unblock_task(NODE_PTR.TASK_PTR, TIME_OUT);
            else
                write(NODE_PTR.TAG, DEQUEUED); -- The time out attempt is over.
            end if;
        when WAIT => -- We must wake up.
            write(NODE_PTR.TAG, DEQUEUED); -- The time out attempt is over.
            timed_unblock_task(NODE_PTR.TASK_PTR, TIME_OUT);
        end case;
    end time_dequeue;
```

-- Code to Disable Time Outs.

```
    procedure disable_to(TIME_NODE)  is
    -- Called to disable a time out when a task that was waiting on a delay
    -- is unblocked to engage in a rendezvous.
    begin
        -- Try to disable the time out.
        case fetch_and_store(TIME_NODE.TAG, REMOVED) is
        when TIME_OUT =>
            -- Time out is still in progress.
            while read(TIME_NODE.TAG) /= DEQUEUED loop  -- Wait for time out to finish.
            -- Note, that the time out will not be interrupted.
            end loop;
        when DEQUEUED =>
            write(TIME_NODE.TAG, DEQUEUED); -- Node can be reused.
        others => -- Left in chain as place holder.
        null;
    end case;
end disable_to;

function time_remove(TIME_NODE) return boolean is
-- Called by the task itself or the aborter when a task waiting at a simple
-- delay statement is aborted.  Note that the node will never be used again.
begin
    return fetch_and_store(TIME_NODE.TAG, REMOVED)= WAIT;
end time_ remove;
```

4.6 Rendezvous Management

The designers of Ada intended that the usual means for tasks to synchronize is via *rendezvous* which are synchronous remote procedure calls. A rendezvous takes place when one task (the *caller*) calls an *entry* of another task, and the task which contains an *accept* statement for the entry (the *owner*) reaches the accept. At this time the statements associated with the accept are executed. At the end of the execution of these statements both of the tasks resume execution. Accept statements can be parameterized, and thus, provide a succinct mechanism for two tasks to synchronize and communicate.

The caller and owner of a entry may optionally provide alternatives to be taken if the rendezvous cannot occur within a certain time – immediately in the case of an else alternative, in a specified amount of time in the case of a delay alternative, and indefinitely in the case of terminate alternative. Despites its complicated interaction with other language features (e.g., termination waves, delays and aborts), we show in this section that the rendezvous does lend itself to a highly parallel implementation.

Nevertheless, the usefulness of rendezvous on highly parallel machines where it may be necessary to synchronize large numbers of tasks may be limited, since synchronizing the tasks two at a time will result in a serial bottleneck. Therefore, shared variables may very well turn out to be the usual means for synchronizing tasks on this class of machines (although this was not the intention of the language designers).

There may, however, be certain tasking idioms whose rendezvous can be executed without serialization on a given machine; for instance, [Schonberg and Schonberg 1985] define a tasking idiom, the beacon task, whose rendezvous can be transformed into fetch&ϕ's on a component of a shared record. On architectures where fetch&ϕ's are combined in the network, these rendezvous will not result in serial bottlenecks even when a large number of tasks are involved. We consider this rendezvous translation scheme in detail in Chapter 5.

There are two Ada attributes, one defined for tasks and one for entries, that concern rendezvous. The attribute defined for tasks is [ARM 9.9 (2)] :

T'CALLABLE	*Yields the value FALSE when the execution of the task designated by T is either completed or terminated, or when the task is abnormal. Yields the value TRUE otherwise. The value of this attribute is of the predefined type BOOLEAN.*

The attribute defined for entries is [ARM 9.9 (6)]:

E'COUNT	*Yields the number of entry calls presently queued on the entry E (if the attribute is evaluated by the execution of an accept statement for the entry, the count does not include the calling task). The value of this attribute is of the type universal integer.*

Accept statements can also be used to implement hardware interrupt handlers and device processes: by associating the address of a device that can cause an interrupt with an entry, the interrupt acts as an entry call issued by the device. Thus, the code to call the interrupt entry is in essence the interrupt handler for the device and the accept statement is the device process. In the next three subsections, we describe accept statements, entry calls, and interrupt entries in more detail.

To implement interrupt entries, SMARTS obviously needs to interact with its underlying architecture. For instance, to allow interrupts issued by a device to behave as entry calls, SMARTS uses of the kernel function *install*. *install(device, handler)* installs *handler* as the interrupt handler for *device*, that is, as the routine to be executed (without software intervention) when *device* issues an interrupt. (A similar function is provided by most operating systems. For example, the UNIX system call *signal*.) Throughout the execution of these routines, hardware interrupts need to be masked. Thus, the kernel functions *mask_interrupt* and *unmask_interrupt* are used in the implementation of interrupts. Interrupts must be masked using these functions during all entry queue operations when preemptive scheduling or interrupt entries are employed by an Ada program.

Finally, since an entry call, the beginning of accept statement, and the end of accept statement are all shared variable synchronization points, we also make use of the kernel functions for flushing the cache of *marked* data (see §4.8.1 and §4.82) in the implementation of rendezvous.

4.6.1 Accept Statements

An accept statement can be standalone or an alternative in a select statement. A task that executes a simple accept cannot proceed until the entry is called. A select statement containing accept alternatives allows a task to accept one of several entries. Associated with each accept is a guard. A select alternative is *open* if either it has no guard or the guard is true. A select that contains an accept statement can have three other types of (mutually exclusive) alternatives: an else clause, a delay or a terminate statement. The else clause of a select specifies what statements should be executed if there are no tasks that have called any of the open entries. The delay alternatives of a select specify how long the task should wait for one of the open entries to be called (before continuing execution). A select with a terminate alternative allows a task waiting for an entry call to terminate when no tasks exist that can call one of the entries of the task (see §4.4).

The ARM specifies when two tasks can achieve rendezvous 9.7 (6-10); we paraphrase these conditions here.

Open alternatives for accept statements are considered first. Selection of an accept statement alternative takes place immediately if the corresponding rendezvous is possible, that is, if there is a pending entry call issued by another task. If several alternatives can thus be selected, one of them is selected arbitrarily. When such an alternative is selected, the corresponding accept statement is executed. If no rendezvous is immediately possible and there is no else part, then the task waits until an open alternative can be selected, i.e., until one of its open entries are called

Selection of other forms of alternatives or of an else part is performed as follows:

- An open delay alternative will be selected if no accept alternative can be selected before the specified delay has elapsed
- The else part is selected and its statements are executed if no accept statement can be immediately selected, in particular, if all alternatives are closed
- An open terminate alternative is selected if no tasks exist which can call one of the entries.

Below we give some examples of accept statements.

Example 1 – Simple Accept

```
accept SYNCHRONIZE;
```

Example 2 – Selective Wait

```
loop -- Process commands to an IO device.
    accept IO_COMMAND(COMMAND : in COMMAND_TYPE; POSITION : ADDRESS;
                      LENGTH : POSITIVE; BUFFER : in out STRING) do
        -- Start a read or write operation.
        ...
    end;
    select
        when READ =>
            accept DONE_READ(DATA : in STRING) do
            -- Read operation complete.
                ...
            end;
    or
        when WRITE =>
            accept DONE_WRITE do
            -- Write operation complete.
                ...
            end,
    or
        delay 5.0;
        - We timed out  so restart operation.
            ...
    end select.
en loop;
```

Example 3 – Selective Nowait

```
loop    Empty a work queue
    select
        accept work queue(JOB : JOB TYPE) do

        end
        else
            -- when no more work exit.
            exit;
        end select;
    end loop;
```

Example 4 – Selective Terminate

```
    task body timer_handler is
        ...
    begin
        loop
            select
                accept TIME_OUT do
                -- Handle the time out.
                    ...
                end;
            or
                -- When we cannot be called terminate.
                terminate;
            end select;
        end loop;
    end timer_handler;
```

4.6.2 Entry Calls

There are three types of entry calls: simple entry calls, conditional entry calls, and timed entry calls. A task that executes a simple entry call must wait unconditionally for the rendezvous to complete before continuing. If the rendezvous is not immediately possible the calling task is queued. (The tasks queued at an entry will be accepted in the order of their arrival.) A conditional entry call, on the other hand, specifies what statements should be executed if the rendezvous is not immediately possible; the entry call is canceled if it is not immediately possible. A timed entry call specifies a delay; the entry call is canceled if it has not been accepted within this delay. Below we give some examples of entry calls.

Example 1 - Simple Entry Call

```
T.SYNCHRONIZE; -- Block until synchronized with a task T.
```

Example 2 - Conditional Entry Call

```
    loop
        select
            T.SYNCHRONIZE;
            return;
        else
            null; -- Busy wait until synchronized with a task T.
        end  select;
    end loop;
```

Example 3 - Timed Entry Call

```
    select
        DRIVER.IO_COMMAND(READ, POSITION, LENGTH, BUFFER);
    or
        delay 10.0;
        -- Device too busy, try again later.
    end select;
```

4.6.3 Interrupt Entries

By associating the address of a device with an entry, interrupts issued by the device act as calls to the entry. In essence, the interrupt handler for the device calls the entry. Tasks can also call interrupt entries. The priority level of this interrupt handler is defined to be higher than that of any user-defined task. When a handler calls an interrupt entry, an implementation is allowed to execute the corresponding accept statement directly without intervention from the scheduler, thereby allowing the body of the accept statement to specify the actions of the handler. Unfortunately, it is not always possible to have the hardware execute an accept statement for an interrupt entry directly. For example, the owner of the interrupt entry may not be in a position to accept the interrupt entry call when it is issued. This optimization is further complicated by having to differentiate between calls made by Ada tasks (which require scheduler intervention) and calls made by the device.

Nevertheless, it should not be too difficult for a compiler to recognize interrupt handler tasks whose bodies consist solely of an infinite loop containing a select statement with an accept for an interrupt entry, and possibly a terminate, as its alternatives (such as the task *timer_handler* described in §4.5.4). If it can also be determined that no Ada tasks call the entry, then there is no problem in having the hardware execute the accept statement directly when an interrupt entry call is generated.

The data structures used to implement accept statements and entry calls are described in the next subsection.

4.6.4 Data Structures for Rendezvous Management

The elements of the *TCB* used specifically for rendezvous management are an array of counters *call_priority*, an array of entry queues *entry*, a pointer *entry_node_ptr*, an array of entry numbers *open* and a flag *rdv*. *call_priority* has an element for each priority level. Each element is the current number of callers of the entries owned by a task with that priority level. (*call_priority* enables rendezvous to be executed at the maximum allowable priority.) The *entry* queues of a task (one for each entry of the task) contain the data structures necessary to manage a parallel accessed queue of *entry_nodes* (these queues are implemented using fetch&stores – see §4.1). These data structures are given in table 4.5.1.

Element	Use
owner_entry	A/B lock used to resolve owner/caller conflicts.
count	The number of tasks queued on the entry.
first	The first task queued on the entry.
guard	The guard of the entry.
last	The last task queue on the entry.

Table 4.5.1 - Entry Queue Data Structures

The *entry_node_ptr* of the *TCB* of a task points to the current *entry_node* of the task. *Open* is used while processing a select statement containing accept alternatives. It contains the entries of the task whose guards are currently open. The *rdv* flag of a task is used to enable and disable rendezvous with the entries owned by the task. *rdv* flags and *entry* queues are discussed in more detail in next subsection.

(SMARTS uses the same mechanism as to compute absolute entry numbers for tasks that are created and activated in the same context The calculation uses the *template_base* and *template_offset* of the *TCB* Thus, these fields are also involved in rendezvous management.

As this mechanism did not have to be altered for SMARTS, we do not describe it here. Interested readers are referred to [Rosen 1985].)

Because rendezvous interact with so many other tasking features, many other element of the *TCB* are peripherally involved in their implementation. These elements include: the *status*, the *event*, the *exception* and the *abnormal* flag. The *status* and *event* of a task are used to coordinate rendezvous. The *status* of a task *t* is inspected by other tasks to determine the current activity of *t* (e.g., *timed_select*). The *event* of a task *t* is set by other tasks to let *t* know what event it is committed to (e.g., *rdv_event*). The *exception* of a task is used to propagate exceptions raised during a rendezvous and to raise TASKING_ERROR in a task that calls a non-callable task. Since rendezvous are abort synchronization points, i.e., abnormal tasks cannot engage in rendezvous, the *abnormal* flag of a task is also used in the implementation of rendezvous.

In order to have a uniform interface to the rendezvous routines, devices associated with interrupt entries need some of the same data structures to manage rendezvous as do task (since the devices can call the entry). Specifically, they need *exception*, *num_events*, *priority* and *save_priority* fields. In addition, if interrupt entries are queued when not handled immediately, then devices need an *entry_node*.

4.6.4.1 Rdv Flags and Entry Queues

Once the owner sets its *rdv* flag, no event involving the owner can happen without resetting the flag. This includes rendezvous discovered by the owner, rendezvous discovered by the caller, a termination wave, a time out and an abort. Whatever actor succeeds in resetting the *rdv* flag has committed the owner to engage in an event with that actor. Termination waves, time outs and aborts are discussed §4.4, §4.5 and §4.7 (respectively).

Similarly, once the caller of an entry enqueues its *entry__node* on an entry queue, no event involving the caller can happen without dequeuing or removing the *entry__node* from the entry queue. The possible events are rendezvous discovered by the owner, a time out or an abort. (The owner of an entry queue will attempt to dequeue *entry__node*s; time outs and aborts will attempt to remove *entry__nodes*.) Whatever actor succeeds in dequeuing or removing the caller from the entry queue has committed the caller to engage in an event with that actor.

Entry queues are implemented as a linked list using fetch&stores as described in §3.2.2.1. In addition to a *first* pointer, a *last* pointer, an A/B-lock *owner__entry*, each entry queue also has a *guard* flag and a *counter*. The *guard* flag is set to **true** by the task that owns the entry when the task reaches a select with open alternative for the entry. The *guard* is reset to **false** by the owner when rendezvous takes place or the rendezvous is canceled (because of a time out). (An array *open* of open entries is maintained to facilitate iterating through the open entries.) The *counter* is the current number of tasks that are waiting on the entry queue. (This *counter* is incremented by enqueuers and decremented by both dequeuers and removers.)

Each entry queue is protected by a non-pre-emptable A/B-lock, *owner__entry*. The owner has priority over the callers, and the callers can access the queue concurrently. In the linked list algorithm given in §3.2.2.1, when a queue was about to empty the dequeuer obtained the B-lock. Because enqueuers are allowed to initiate rendezvous the dequeuer (e.g., owner) always obtains the A-lock. We allow dequeuers to initiate rendezvous so that rendezvous take place as soon as possible. This also prevents the dequeuer from missing an enqueue that takes place after it has inspected an entry queue. To avoid race conditions, the owner of an entry must obtain the A-lock when the *guard* is set or reset and the caller of an entry must obtain the B-lock to either enqueue or remove a node (see §4 6 7).

After the owner of an entry queue actually dequeues *entry__nodes* that have been effectively removed from (but left as place holders in) one of its entry queues Since no other actor will attempt to access such *entry__nodes* it is safe for their owners to reuse them.

4.6.5 Execution of a Select Statement

When the owner of entry wishes to engage in a rendezvous, if its *abnormal* flag is not set, then the owner sets its *rdv* flag to **true**. The owner then loops through the open entries trying to find a non-empty entry queue as follows: First the owner obtains an A-lock for the entry; it will keep this lock while inspecting the queue. After which the *counter* of the entry queue is decremented with a fetch&add.

If a zero value is returned from the fetch&add, then the *guard* for the entry is set to **true** – from this point on a rendezvous can be initiated by callers the entry. The owner releases the A-lock for the entry and moves on to the next open entry.

If a non-zero value is returned from the fetch&add, then the owner has reserved one of the nodes on the queue (no node can be removed while the A-lock is kept), so it will attempt to reset the *rdv* flag with a fetch&store(*owner rdv*, **false**) = **true**. If the owner also succeeds in resetting the *rdv* flag, then it has committed itself to a rendezvous with the first (non-removed) task in the entry queue.

It then sets its *status* (to the type of select it is executing) – the owner can now be unblocked by an aborter

If the owner does not find a non empty (open) entry queue or the *rdv* flag is set to **false** by either an aborter or a caller, then what the owner does next depends on whether the select was conditional or nonconditional:

If the select was conditional, i e , it contains an else alternative, then the owner attempts to cancel the rendezvous by disabling the *rdv* flag If the attempt to reset the *rdv* flag is successful it continues executing. Otherwise the owner awaits the necessarily pending entry call. If a pre-emptive scheduling policy or interrupt entries are in use that the owner executes a *block__task* - it will be unblocked by the caller that succeeded in disabling the *rdv* flag. If neither is in use then the owner can execute a *busy__wait__owner* obviating its context switch.

If the select is not conditional, then the owner sets its *status* to the type of select it is executing - the owner can now be unblocked by an aborter. Thus, the owner inspects its *abnormal* flag to check if it has already been aborted. (The owner must check its *abnormal* flag before and after setting its *status* to avoid a race condition - see §4.1.1.)

If the owner has been aborted, it tries to reset the rendezvous. If the owner is not successful at disabling the rendezvous, i.e., a caller or an aborter has reset the *rdv* flag, then the completion of the abort must be postponed; if a caller has restted the *rdv* flag, then the abort will be completed after the rendezvous; if the aborter has resetted the *rdv* flag the abort will be completed after the owner is unblocked.

If the owner has not been aborted or is unsuccessful at disabling the rendezvous, then the owner will (eventually) block. The actions performed before the owner blocks will be determined by what type of alternatives the select contains.

If the select has only accept alternatives, then the owner executes a *block__task* (we treat simple accepts as selects with only one alternative).

If the select contains a delay alternative, then the owner enables a time out for the delay (see §4.5) and then executes a *timed__block__task* - it will be unblocked by either an aborter, a caller or a time out.

If the select contains a terminate alternative, then the owner checks if it is quiescent (see §4.4) and then executes a *block__task* - it will be unblocked by either an aborter, a caller or a termination wave.

Before the owner is unblocked, its *event* will have been set by the actor that reset the *rdv* flag of the owner, and hence, the owner will know what event it has been committed to. (See §4.2 for an explanation of task blocking and unblocking.) Upon being unblocked, an owner will set its *status* to *active* (thereby, disabling abort events).

4.6.6 Execution of an Accept

The start and end of an accept statement is a shared variable synchronization point; therefore, synchronous shared variables must be flushed from the cache (see §4.8). The start and end of an accept statement are also abort synchronization points; therefore, the *abnormal* flags of the owner and caller must be checked.

Prior to executing the actual statements associated with an accept, i.e. the rendezvous, the open *guards* on the *entry* queues of the owner are closed (note that the A-lock's of an entry must be held while its *guard* is being updated), and the caller is linked onto the *serviced* tasks chain of the owner. This chain is used to raise the exception TASKING_ERROR in all the tasks that the owner is currently engaged in rendezvous with, should the owner be aborted (see §4.7).

Accept statements must be executed the maximum priority level of the owner and caller if both are defined. If the priority of the owner is undefined then the accept is executed at the maximum priority level of any task waiting an entry queue of the owner. Thus, the priority of the owner is saved in *caller.save_priority* (We cannot save the priority in *owner.save_priority* since we may have nested accept statements). If the priority of the owner is undefined, then *owner.priority* is set to the maximum index of the array *owner.call_priority* whose element is non-zero. (When a caller *c* places itself on an entry queue of an owner *o*, it increments *o.call_priority(c.priority)*.) Otherwise, *owner.priority* is set to the maximum of the priority level of the owner and caller. At the completion of the accept statement *owner.priority* is restored (from *caller.save_priority*) and the caller is removed from the *serviced* chain of the owner.

Finally, if an (unhandled) exception occurred during the rendezvous it is propagated to the caller via its *exception* field.

4.6.7 Execution of an Entry Call

After obtaining a B-lock of the A/B-lock *owner__entry* for the entry, an entry caller sets its *status* to whatever type of entry call it is executing (a conditional call sets its *status* to *call*). The caller then inspects its *abnormal* flag. If the caller has been aborted then it releases the B-lock and completes the abort. Before engaging in a rendezvous with the owner of the entry, the caller must also ensure that the owner is still callable (i.e., not abnormal) complete or terminated. If the owner is not callable, then the caller releases the b-lock and raises the exception TASKING__ERROR.

Otherwise the caller attempts to initiate a rendezvous with the owner of the entry as follows: It increments the *counter* for the entry with a fetch&add(*entry.counter*, 1). If the value returned from the fetch&add is zero, then the caller knows it is the first task on the entry queue, so it then inspects the *guard* flag. If the *guard* flag is **true**, then the caller attempts to reset the the *rdv* flag of the owner with a fetch&store(*owner.rdv*, **false**) = **true**. If the caller succeeds in this, then the owner is committed a rendezvous with the caller. Hence, the caller releases the B-lock and unblocks the owner with a *start__rdv* event.

If the caller is not able to engage in a rendezvous immediately, then what it does next depends whether it is executing a conditional, timed, or simple entry call:

If the call is conditional then, after incrementing *counter* with a *fetch&add*(*entry.counter*, 1), the caller must check if it has been aborted, i.e., its *abnormal* flag is set.. (The *abnormal* flag must be re-inspected to avoid the race condition discussed in §7.1 1..) If the caller was aborted, then it sets its *status* to *complete*, releases the B-lock, and completes the abort. If the caller was not aborted , then it sets its *status* to *active*, releases the B-lock, and executes the else condition.

If the call is not conditional, then the caller enqueues itself on the entry queue and increments the element of the owner's *call__priority* for the priority level of the caller. (Note that the element of the owner's *call__priority* will be decremented when the caller is removed from the entry queue.) After placing itself on the entry queue, the caller inspects its *abnormal* flag. If the caller has been aborted, then it tries to remove itself from the queue. If the caller is successful at removing itself, then it releases the B-lock, sets its *status* to *complete*, and completes the abort. Otherwise the caller sets its *status* to *active*, releases the B-lock and blocks. The caller will be unblocked by whoever removed it from the entry queue – the owner or an aborter.

If the call is a timed entry call, then the caller must also enqueue itself on the time chain of the SMARTS processor (that is currently executing the caller) before executing a *timed__block__task*. If the call is a simple entry call, then the caller executes a *block__task*.

Before the caller is unblocked, its *event* will have been set by the actor that removed the caller from the entry queue. Hence, the caller will know what event has occurred. After the caller is unblocked, it sets its status to *active* (thereby, disabling abort events).

4.6.8 Execution of Interrupt Entry Calls

Interrupt entries can be called by both tasks and hardware generated interrupts. Furthermore, accept statements for interrupt entries can appear in select statements wherever accept statements for non interrupt entries can. For compatibility, we use the same routines for the accept statements of interrupt entries as we do for accept statement for non-interrupt entries.

To allow hardware interrupts generated by a device to interface to the accept routines, we provide two entry routines which can be installed as the interrupt handler for the device. These entry routines are similar to the routines for simple and conditional entry calls. (A simple interrupt entry call specifies that pending interrupts are to queued and handled later; a conditional entry interrupt call specifies that interrupts are to be ignored if not handled immediately.)

There are several differences between the interrupt and non-interrupt entry routines: The interrupt entry routines must disable hardware interrupts during the entire routine (not just during the entry queue operations). Furthermore, if the rendezvous can be begun immediately and the owner is blocked they will force a context switch to the owner of the entry (given that software interrupts are not disabled). In addition, the interrupt entry routines need not check if the interrupt entry caller is aborted. A slight complication is that after a rendezvous initiated by a hardware interrupt is completed, there is no caller to unblock. By associating a *num__events* counter with the interrupt and setting this *num__event* to 1 (rather than 0), however, we ensure that no attempt will be made to unblock the (nonexistent) interrupt entry caller.

4.6.9 Code for Rendezvous

-- Code for Actions of the Owner.

```
procedure select_nowait(OWNER) is
-- Execute a conditional select statement.
begin
    if read(OWNER.ABNORMAL) /= 0 then -- This is an abort synchronization point.
        abortme(OWNER, NOT_TERM);
    end if;
    fetch_and_store(OWNER.RDV, true); -- A rendezvous is now possible.
    if owner_rdv(OWNER, CALLER, ENTRY) then
        start_accept(OWNER, ENTRY, CALLER);
    else
        write(OWNER.STATUS, SELECT);
        if fetch_and_store(OWNER.RDV, false) = true then
        -- The rendezvous is disabled
            if read(OWNER.ABNORMAL) /= 0 then -- We were aborted.
```

```
               abortme(OWNER, NOT_TERM);
            else
               write(OWNER.STATUS, ACTIVE);
            end if;
         else -- We were aborted or a rendezvous is under way.
            -- **If pre-emptive scheduling use wait_owner.
            busy_wait_owner(OWNER); -- No rendezvous possible or one started.
         end if;
      end if;
   end select_nowait;

   procedure select(OWNER) is
   -- Execute a select statement with only accept alternatives.
   begin
      if read(OWNER.ABNORMAL) /= 0 then -- This is a synchronization point.
         abortme(OWNER, NOT_TERM);
      end if;
      fetch_and_store(OWNER.RDV, true); -- A rendezvous is now possible.
      if owner_rdv(OWNER, CALLER, ENTRY) then
         start_accept(OWNER, ENTRY, CALLER);
         return;
      end if;
      write(OWNER.STATUS, SELECT);
      if OWNER.ABNORMAL /= 0 and then  fetch_and_store(OWNER.RDV, false) = true then
      -- We were aborted and the rendezvous is disabled.
         abortme(OWNER, NOT_TERM);
      else -- Wait for rendezvous or abort.
         wait_owner(OWNER);
      end if;
   end select;

   procedure select_term(OWNER) is
   -- Execute a select statement with only accept alternatives.
   begin
      if read(OWNER.ABNORMAL) /= 0 then -- This is an abort synchronization point.
         abortme(OWNER, NOT_TERM);
      end if;
      fetch_and_store(OWNER.RDV, true); -- A rendezvous is now possible.
      if owner_rdv(OWNER, CALLER, ENTRY) then
         start_accept(OWNER, ENTRY, CALLER);
         return;
      end if;
      write(OWNER.STATUS, TERM_SELECT); -- Abort can now succeed.
      if read(OWNER.ABNORMAL) /= 0 and then fetch_and_store(OWNER.RDV, false) = true
      then
      -- We have been aborted and the rendezvous is disabled.
         abortme(OWNER, NOT_TERM);
      else
          if fetch_and_add(OWNER.NO_TERM, -1) = 1 then
          -- We are quiescent, so check our masters.
          -- A termination wave is now possible.
             check_term(OWNER.MASTER_TASK, OWNER.MASTER_BLOCK);
         end if;
```

```
        term_wait_owner(OWNER); -- Wait for rendezvous or termination.
    end if;
end select_term;

procedure select_wait(OWNER; DELAY) is
-- Execute a select statement with accept and delay alternatives.
begin
    if read(OWNER.ABNORMAL) /= 0 then -- This is a synchronization point.
        abortme(OWNER, NOT_TERM);
    end if;
    fetch_and_store(OWNER.RDV, true); -- A rendezvous is now possible.
    if owner_rdv(OWNER, CALLER, ENTRY) then
        start_accept(OWNER, ENTRY, CALLER);
        return;
    end if;
    write(OWNER.STATUS, TIMED_SELECT); -- Aborts can now succeed.
    if OWNER.ABNORMAL /= 0 and then fetch_and_store(OWNER.RDV, false) = true then
    -- We were aborted and the rendezvous is disabled.
        abortme(OWNER, NOT_TERM);
    end if;
    time_enqueue(OWNER, DELAY, TIMED_SELECT); -- Enable Time Out.
    timed_wait_owner(OWNER, DELAY); -- No rendezvous possible or one started.
end select_wait;

function owner_rdv(OWNER; CALLER; ENTRY) return boolean is
-- Check if OWNER can initiate a rendezvous.
begin
    -- See if we can rendezvous.
    for each I in read(OWNER.OPEN) loop
        if read(OWNER.RDV) = false then -- Set by caller or aborter.
            return false;
        end if;
        nonpreempt_a_lock(OWNER.ENTRY(I).OWNER_CALLER); -- Get the entry lock.
        COUNT := fetch_and_add(OWNER.ENTRY(I).COUNTER, -1);
        if COUNT = 0 or else fetch_and_store(OWNER.RDV, false) = false then
            -- No task waiting or we didn't get the flag.
            add(ENTRY(I).COUNTER, 1);
            write(OWNER.ENTRY(I)].GUARD, true);
            nonpreempt_a_unlock(OWNER.ENTRY(I).OWNER_CALLER);
        else -- A task is waiting and we got the flag, so a rendezvous is possible.
            CALLER := entry_dequeue(OWNER.ENTRY(I));
            nonpreempt_a_unlock(OWNER.ENTRY(I).OWNER_CALLER);
            return true;
        end if;
    end loop;
    return false;
end owner_rdv;

procedure busy_wait_owner(OWNER)  is
-- The owner is waiting on a no-wait select statement whose event is imminent.
-- **Do not use with pre-emptive scheduling!
begin
    busy_wait(OWNER); -- Busy wait for an event.
    write(OWNER.STATUS, ACTIVE);
    case read(OWNER.EVENT) of -- Each event has disabled the others.
    when START_RDV => -- We are committed to rendezvous.
        start_accept(OWNER, OWNER.WHAT, OWNER.WHO);
    when ABORT => -- We were aborted.
        abortme(OWNER, NOT_TERM); -- False because we have not checked quiescence.
    end case;
end busy_wait_owner;

procedure wait_owner(OWNER)  is
-- The owner is waiting on a simple select statement.
begin
    block_task(OWNER); -- Wait for an event.
    write(OWNER.STATUS, ACTIVE);
    case read(OWNER.EVENT) of -- Each event has disabled the others.
    when START_RDV => -- We are committed to rendezvous.
        start_accept(OWNER, OWNER.WHAT, OWNER.WHO);
    when ABORT => -- We were aborted.
        abortme(OWNER, NOT_TERM); -- False because we have not checked quiescence.
    end case;
end wait_owner;

procedure term_wait_owner(OWNER) is
-- The owner is waiting on a select with a terminate alternative.
begin
    block_task(OWNER); -- Wait for an event.
    write(OWNER.STATUS, ACTIVE);
    case read(OWNER.EVENT) of -- Each event has disabled the others.
    when START_RDV => -- We are committed to rendezvous.
        check_unterm(OWNER); -- We are no longer quiescent.
        start_accept(OWNER, OWNER.WHAT, OWNER.WHO);
    when TERMINATE =>
        terminateme(OWNER);
    when ABORT =>
        abortme(OWNER, TERM);
    end case;
end term_wait_owner;

procedure timed_wait_owner(OWNER) is
-- The owner is waiting on a select statement with a delay alternative
begin
    timed_block_task(OWNER); -- Wait for an event
    write(OWNER.STATUS, ACTIVE);
    case read(OWNER.EVENT) of --  Each event has disabled the others
    when START_RDV => -- We are committed to rendezvous.
        disable_to(OWNER.TIME_NODE_PTR);
        start_accept(OWNER, OWNER.WHAT, OWNER.WHO);
    when TIME_OUT => -- This is a synchronization point.
        if OWNER.ABNORMAL /= 0 then -- We have been aborted.
            abortme(OWNER, NOT_TERM);
        end if;
    when ABORT =>
        disable_to(OWNER.TIME_NODE_PTR);
        abortme(OWNER, NOT_TERM);
    end case;
end timed_wait_owner;
```

-- Code for the Accept.

```
procedure start_accept(OWNER; ENTRY; CALLER) is
-- Execute accept statement ENTRY, for rendezvous between OWNER and CALLER
begin
    -- Bump our priority as soon as possible.
    write(CALLER.SAVE_PRIORITY,  OWNER.PRIORITY); -- Save owners priority.
    if read(OWNER.PRIORITY) = 0 then -- I.e., Owners priority undefined.
    -- Execute at the maximum priority of all callers.
        for PRIORITY in reverse MAX_PRIORITY..1 loop
            if read(OWNER.CALL_PRIORITY(I)) /= 0 then
                HIGHEST_PRIORITY := PRIORITY;
                exit;
            end if;
        end loop;
    else
        HIGHEST_PRIORITY := max(OWNER.PRIORITY, CALLER.PRIORITY);
    end if;
    write(OWNER.PRIORITY, HIGHEST_PRIORITY);
    flush(volatile); -- This is a shared variable synchronization point.
    -- Add caller to the serviced to_enqueue of the owner
    -- (needed if owner is aborted to raise exceptions).
    write(CALLER.NEXT, OWNER.SERVICED);
    write(OWNER.SERVICED, CALLER);

    -- Execute the statements associated with the accept ...

end start_accept;

procedure end_accept(OWNER; CALLER) is
-- Rendezvous between OWNER and CALLER is over.
begin
    -- Propagate any exception that was raised to calling task.
    write(CALLER.EXCEPTION, OWNER.EXCEPTION);
    if read(OWNER.ABNORMAL) /= 0 then -- Owner aborted, raise TASKING_ERROR in
        write(CALLER.EXCEPTION) := TASKING_ERROR;
    end if;
    flush(volatile);  -- This is a shared variable synchronization point.
    -- Restore owners priority.
    write(OWNER.PRIORITY, CALLER.SAVE_PRIORITY);
    -- Unblock CALLER with an END_RDV event (It will discover TASKING_ERROR.).
    if read(CALLER.STATUS) = TIMED_CALL then
        timed_unblock_task(CALLER, END_RDV);
    else
        unblock_task(CALLER, END_RDV);
    end if;
    if read(OWNER.ABNORMAL) /= 0 then  -- This is an abort synchronization point.
        abortme(OWNER, NOT_TERM);
    end if;
    -- Clean up.
    write(OWNER.SERVICED, OWNER.SERVICED.NEXT);
```

```
        close_guards(OWNER);
        add(OWNER.CALL_PRIORITY(CALLER.PRIORITY), -1);
    end end_accept;

    procedure close_guards(OWNER) is
    -- Close the guards of a task after a rendezvous has been started or disabled.
    begin
        for I in read(OWNER.OPEN) loop
            nonpreempt_a_lock(OWNER.ENTRY(I).OWNER_CALLER);
            write(OWNER.ENTRY(I).GUARD, false);
            nonpreempt_a_unlock(OWNER.ENTRY(I).OWNER_CALLER);
        end loop;
    end close_guards;
```

-- Code for Actions of the Caller.

```
    procedure entry_call(CALLER; ENTRY; OWNER)  is
    -- CALLER is executing a simple entry call.
    begin
        nonpreempt_b_lock(ENTRY.OWNER_ENTRY); -- Entering critical section.
        write(CALLER.STATUS, CALL);    -- Must set so that aborts are not missed.
        if read(CALLER.ABNORMAL) /= 0 then -- This is an abort synchronization point.
            write(STATUS, COMPLETED);
            nonpreempt_b_unlock(ENTRY.OWNER_ENTRY); -- Exiting critical section.
            abortme(CALLER, NOT_TERM);
        end if;
        if not is_callable(OWNER) then
            write(CALLER.STATUS, ACTIVE);
            nonpreempt_b_unlock(ENTRY.OWNER_ENTRY); -- Exiting critical section.
            raise(TASKING_ERROR);
        end if;
        -- Try to initiate the rendezvous.
        if not caller_rdv(CALLER, ENTRY, OWNER) then
            -- Indicate that a task of caller's priority is waiting.
            add(OWNER.CALL_PRIORITY(CALLER.PRIORITY), 1);
            flush(volatile); -- This is a shared variable synchronization point.
            write(CALLER.WHAT, ENTRY);
            entry_enqueue(CALLER.ENTRY_NODE_PTR, ENTRY.QUEUE); -- Aborts can now succeed.
            if read(CALLER.ABNORMAL) /= 0 then
            -- We have been aborted.
                if entry_remove(CALLER) then
                -- The entry call is canceled.
                    write(CALLER.STATUS, COMPLETED);
                    nonpreempt_b_unlock(ENTRY.OWNER_CALLER); -- Exiting critical section.
                    abortme(CALLER, NOT_TERM);
                else
                -- Rendezvous or abort already started
                    write(CALLER.STATUS, ACTIVE); -- Let aborter continue
                end if;

            end if; -- Caller_rdv released B lock
            wait_caller(CALLER);
        end if;
```

```
end entry_call;

procedure conditional_call(CALLER; ENTRY; OWNER)  is
-- CALLER is executing a conditional entry call
begin
    nonpreempt_b_lock(ENTRY.OWNER_ENTRY); -- Entering critical section.
    write(CALLER.STATUS, CALL);   -- Must set so that aborts are not missed.
    if read(CALLER.ABNORMAL) /= 0 then -- This is an abort synchronization point.
        write(STATUS, COMPLETED);
        nonpreempt_b_unlock(ENTRY.OWNER_ENTRY); -- Exiting critical section.
        abortme(CALLER, NOT_TERM);
    end if;
    if not is_callable(OWNER) then
        write(CALLER.STATUS, ACTIVE);
        nonpreempt_b_unlock(ENTRY.OWNER_ENTRY); -- Exiting critical section.
        raise(TASKING_ERROR);
    end if;
    -- Try to initiate  the rendezvous.
    if not caller_rdv(CALLER, ENTRY, OWNER) then
        if read(CALLER.ABNORMAL) /= 0 then
            write(CALLER.STATUS, COMPLETED);
            nonpreempt_b_unlock(ENTRY.OWNER_CALLER); -- Exiting critical section.
            abortme(CALLER, NOT_TERM); -- We have been aborted.
        else
            write(CALLER.STATUS, ACTIVE);
            nonpreempt_b_unlock(ENTRY.OWNER_CALLER); -- Exiting critical section.
        end if;
    end if; -- Caller_rdv released B lock.
end conditional_call;

procedure timed_call(CALLER; ENTRY; OWNER; DELAY)  is
-- CALLER is executing a entry call with a delay alternative.
begin
    nonpreempt_b_lock(ENTRY.OWNER_ENTRY); -- Entering critical section.
    write(CALLER.STATUS, TIMED_CALL); -- Abort attempts are now enabled.
    if read(CALLER.ABNORMAL) /= 0 then -- This is an abort synchronization point.
        write(CALLER.STATUS, COMPLETED);
        nonpreempt_b_unlock(ENTRY.OWNER_ENTRY); -- Exiting critical section.
        abortme(CALLER, NOT_TERM);
    end if;
    if not is_callable(OWNER) then
        write(CALLER.STATUS, ACTIVE); -- Call attempt over.
        nonpreempt_b_unlock(ENTRY.OWNER_ENTRY); -- Exiting critical section.
        raise(TASKING_ERROR);
    end if;
    -- Try to initiate the rendezvous.
    if not caller_rdv(CALLER, ENTRY, OWNER) then
        -- Indicate that a task of caller's priority is waiting.
        add(OWNER.CALL_PRIORITY(CALLER.PRIORITY), 1);
        flush(volatile); -- This may be a synchronization point.
        write(CALLER.WHAT, ENTRY);
        entry_enqueue(CALLER.ENTRY_NODE_PTR, ENTRY.QUEUE);  -- Aborts can now succeed.
        -- We must wait for rendezvous or abort.
        if read(CALLER.ABNORMAL) /= 0 then
        -- We have been aborted.
            if entry_remove(CALLER) then
            -- The entry call is canceled.
                write(CALLER.STATUS, COMPLETED);
                nonpreempt_b_unlock(ENTRY.OWNER_CALLER); -- Exiting critical section.
                abortme(CALLER, NOT_TERM);
            else
            -- Rendezvous already started.
                write(CALLER.STATUS, ACTIVE); -- Let aborter continue.
            end if;
        and if;
        nonpreempt_b_unlock(ENTRY.OWNER_CALLER);  -- Exiting critical section.

        time_enqueue(CALLER, DELAY, TIMED_CALL);
        timed_wait_caller(CALLER);
    end if; -- Caller_rdv released B lock.
end timed_call;

function caller_rdv(CALLER; ENTRY; OWNER) return boolean is
-- Check if a rendezvous between CALLER and OWNER is immediately possible.
-- Note we have a B-lock.
begin
    if fetch_and_add(ENTRY.COUNTER, 1) = 0 and then ENTRY.GUARD = true
    and then fetch_and_store(OWNER.RDV, false) = true then -- rendezvous possible.
    -- The value of the guard cannot change while I have a nonpreempt_b_lock !!
        fetch_and_add(ENTRY.COUNTER, -1);
        write(CALLER.STATUS, ACTIVE); -- Let aborter continue.
        nonpreempt_b_unlock(ENTRY.OWNER_CALLER); -- Release B-lock.
        write(OWNER.WHO, CALLER);
        write(OWNER.WHAT, ENTRY);
        flush(volatile); -- This is a shared variable synchronization point.
        add(OWNER.CALLER_PRIORITY(CALLER.PRIORITY), 1);
        -- Unblock OWNER with a RDV_EVENT.
        if read(OWNER.STATUS = TIMED_SELECT) then
            timed_unblock_task(OWNER, RDV_EVENT);
        else
            unblock_task(OWNER, RDV_EVENT);
        end if;
        return true;
    else
        -- Note we have not given up B lock.
        return false;
    end if;
end call_rdv;

procedure wait_caller(CALLER) is
-- CALLER is waiting on an entry call.
begin
    block_task(CALLER); -- Wait for an event
    -- The rendezvous is over, the owner was aborted or we were aborted.
    write(CALLER.STATUS, ACTIVE);
    if read(CALLER.ABNORMAL) /= 0 then -- This is an abort synchronization point.
        abortme(CALLER, NOT_TERM);
    end if;
```

```
        if read(CALLER.EXCEPTION) /= NO_EXCEPTION then
            raise(CALLER.EXCEPTION);
        end if;
    end wait_caller;

    procedure timed_wait_caller(CALLER) is
    -- The CALLER is waiting on an entry with a delay alternative.
    begin
        timed_block_task(CALLER); -- Wait for an event.
        -- The rendezvous is over, we timed out, the owner was aborted or we were aborted.
        write(CALLER.STATUS, ACTIVE);
        if read(CALLER.ABNORMAL) /= 0 then -- This is an abort synchronization point.
            abortme(CALLER, NOT_TERM);
        end if;
        if read(CALLER.EVENT) = END_RDV then -- A rendezvous or the owner was aborted.
            disable_to(CALLER.TIME_NODE_PTR);
            if read(CALLER.EXCEPTION) /= NO_EXCEPTION then
                raise(CALLER.EXCEPTION);
            end if;
        end if;
    end timed_wait_caller;
```

-- Interrupt Entries.

```
    procedure interrupt(DEVICE; ENTRY; OWNER) is
    -- A simple entry Interrupt entry call.
    begin
        mask_interrupts; -- We don't want to be interrupted.
        if read(OWNER.ABNORMAL) then
            unmask_interrupts; -- Enable other interrupt handlers.
            return;
        end if;
        if not interrupt_rdv(DEVICE, ENTRY, OWNER) then
        -- Cannot immediately initiate a rendezvous with OWNER.
            -- Ensure that we are not notified of END_RDV event.
            -- Note that DEVICE.STATUS is always set to CALL.
            add(DEVICE.NUM_EVENT, 1);  -- Ensure we are not unblocked.
            enqueue(DEVICE.ENTRY_NODE_PTR, ENTRY);
            b_unlock(ENTRY.OWNER_CALLER); -- Release B lock obtained in interrupt_rdv.
            -- Note that the STATUS of a device is permanently set to CALL.
        end if;
        unmask_interrupts; -- Enable other interrupt handlers.
    end interrupt;

    procedure conditional_interrupt(CALLER; ENTRY; OWNER) is
    -- A conditional interrupt entry call.
    begin
        mask_interrupts; -- We don't want to be interrupted.
        if read(OWNER.ABNORMAL) then
            unmask_interrupts; -- Enable other interrupt handlers.
            return;
        end if;
        -- Interrupt ignored if we can't rendezvous immediately.
        if not interrupt_rdv(DEVICE, ENTRY, OWNER) then
            b_unlock(ENTRY.OWNER_CALLER); -- Release B lock obtained in interrupt_rdv.
        end if;
        unmask_interrupts; -- Enable other interrupt handlers.
    end conditional_interrupt;


    function interrupt_caller_rdv(CALLER; ENTRY; OWNER) return boolean is
    -- Caller is attempting to rendezvous.
    begin
        b_lock(ENTRY.OWNER_CALLER); -- Obtain entry b lock.
        if fetch_and_add(ENTRY.COUNTER, 1) = 0
        and then ENTRY.GUARD = true
        and then fetch_and_store(OWNER.RDV, false) = true then
        -- rendezvous possible.
            add(ENTRY.COUNTER, -1);
            write(OWNER.WHO, CALLER);
            write(OWNER.WHAT, ENTRY);
            fetch_and_add(OWNER.CALLER_PRIORITY(MAX_PRIORITY), 1);
            -- Unblock OWNER with a RDV_EVENT and try to execute.
            if read(OWNER.STATUS = TIMED_SELECT) then
                timed_unblock_and_run(OWNER, RDV_EVENT);
            else
                unblock_and_run(OWNER, RDV_EVENT);
            end if;
            b_unlock(ENTRY.OWNER_CALLER); -- Release entry b lock.
            return true;
        else
            return false; -- Note entry still held.
        end if;
    end interrupt_caller_rdv;
```

-- Code for Non-Pre-emptable Locks.

```
    procedure nonpreempt_a_lock(AB) is
    -- Obtain non-pre-emptable A lock.
    begin
        -- Entering critical section.
        -- Hardware disable pre-emption (only needed for preemptive scheduling).
        mask_interrupts;
        a_lock(AB);
    end nonpreempt_a_lock;

    procedure nonpreempt_a_unlock(AB) is
    -- Release non-pre-emptable B lock.
    begin
        a_unlock(AB);
        -- Leaving critical section.
        -- Hardware enable pre-emption.
        unmask_interrupts;
    end nonpreempt_a_lock;

    procedure nonpreempt_b_lock(AB) is
```

```
-- Obtain non-pre-emptable B lock.
begin
    -- Entering critical section.
    -- Hardware disable pre-emption (only needed for preemptive scheduling).
    mask_interrupts;
    b_lock(AB);
end nonpreempt_b_lock;

procedure nonpreempt_b_unlock(AB) is
-- Release non-pre-emptable B lock.
begin
    b_unlock(AB);
    -- Leaving critical section.
    -- Hardware enable pre-emption (only needed for pre-emptive scheduling.
    unmask_interrupts;
end nonpreempt_b_lock;
```

-- Entry Queue Operations.

```
procedure entry_enqueue(CALLER; ENTRY) is
-- Enqueue a CALLER.ENTRY_NODE_PTR on ENTRY queue.
-- Note: we already have a caller lock.
    begin
        if read(CALLER.ENTRY_NODE_PTR.TAG) /= DEQUEUED then
        -- The node is acting as a place holder in some queue.
            write(CALLER.ENTRY_NODE_PTR, new ENTRY_NODE);
        end if;
        CALLER_NODE := read(CALLER.ENTRY_NODE_PTR);
        write(CALLER_NODE.FLAG, ENQUEUED); -- Reset flag.
        PREVIOUS := fetch_and_store(ENTRY.LAST, CALLER_NODE);
        if PREVIOUS = null then
            write(ENTRY.FIRST, CALLER_NODE);
        else
            write(PREVIOUS.NEXT, CALLER_NODE);
        end if;
end entry_enqueue;

function entry_dequeue(ENTRY)  return TCB_REC_PTR is
-- Dequeue a node from entry queue.
Note: we already have an nonpreempt_a_lock and have reserved a node.
begin
    loop
        CALLER_NODE := fetch_and_store(ENTRY.FIRST, ENTRY.FIRST.NEXT);
        if fetch_and_store(CALLER_NODE.TAG, DEQUEUED) = FREE then
            -- The node has not been removed.
            if read(ENTRY.FIRST) = null then -- QUEUE about to be empty.
                write(ENTRY.LAST, EMPTY);
            end if;
            return CALLER_NODE.TASK;
        end if;
    end loop;
end entry_dequeue;
```

```
function entry_remove(CALLER) return boolean is
-- Remove CALLER from the entry queue it is on.
-- Called only by the owner or aborter.  The aborter will have set CALLER.ABNORMAL,
-- so it will not be put on a different queue.
begin
    ENTRY := read(CALLER.WHAT); -- Entry queue that caller is on.
    nonpreempt_b_lock(ENTRY.AB);
    CALLER_NODE := read(CALLER.ENTRY_NODE_PTR);
    case fetch_and_store(CALLER_NODE.TAG, REMOVED) of
    when ENQUEUED => -- We got the flag, leave node in queue as place holder.
        -- One less task of callers priority waiting.
        add(CALLER.WHO.CALL_PRIORITY(CALLER.PRIORITY), -1);
        add(ENTRY.COUNTER, -1);
        nonpreempt_b_unlock(ENTRY.OWNER_CALLER);
        return true;
    when DEQUEUED =>
        write(N.TAG, DEQUEUED); -- We missed it, but the node can be reused.
        nonpreempt_b_unlock(ENTRY.OWNER_CALLER);
        return false;
    when REMOVED => -- Some other remover got it, and the node is a place holder.
        nonpreempt_b_unlock(ENTRY.OWNER_CALLER);
        return false;
    end case;
end entry_remove;
```

-- Code for Rendezvous Attributes.

```
type STATUS_TYPE is (CREATE, ACTIVE, ACTIVATE, CALL, TIMED_CALL, SELECT, TERM_SELECT,
                     TIMED_SELECT, WAIT, TERMINATE, COMPLETE, TERMINATED);
```

-- Code for CALLABLE.

```
function is_callable(TASK) return boolean is
begin
    -- The task is COMPLETED or TERMINATED or ABORTED.
    return (read(TASK.STATUS) >= COMPLETED) or else (read(TASK.ABNORMAL) /= 0);
end is_callable;
```

-- Code for COUNT.

```
function count(ENTRY) return integer is
begin
    return read(ENTRY.COUNT);
end count;
```

4.6.10 Absence of Race Conditions

It is difficult to be convinced of the lack of race conditions in any rendezvous mechanism without taking into account its interaction with other Ada features, specifically, termination waves, time outs, and aborts. Thus, we briefly discuss the actions performed by the actors (master, timer handler, or aborter) that cause these events here. (For details on termination waves, aborts and time outs §4.4, §4.6, and §4.7, respectively.) We then show that there are no race conditions between the callers and owners of an entry.

Termination Waves

A task *t* waiting on a select with a terminate alternative cannot be involved in a termination wave while there exist tasks that can call *t*. Nor can a task *t* waiting on a select with a terminate alternative be involved in a time out (a select with a terminate alternative cannot have a delay alternative). A task that is waiting on a terminate alternative can, however, be aborted. To avoid a potential with aborters to unblock a task that is waiting on a select with a terminate alternative, the master of the task must reset the *rdv* flag of the task when the master is participating in a termination wave. Only if the *rdv* flag is successfully reset will the master unblock the task with a *terminate* event.

Time Outs

When a tasks executes a delay the *time__node* of the task is enqueued to the time chain of SMARTS processor that is executing the task. When the delay of a *time__node* on the chain of a SMARTS processor expires, if the delay has not been canceled, then the *timer__handler* will also try to reset the *rdv* flag or remove the task from the entry queue. If the time out does not succeed in resetting the *rdv* flag or removing the task, then the time out fails; a rendezvous or an abort has already begun.

Aborts

When a task is aborted its *abnormal* flag is set. All tasks check their *abnormal* flags at abort synchronization points. If the aborted task is executing a *select* the aborter will try to reset the *rdv* flag of that task. If the aborted task is executing an *entry call* the aborter will try to remove that task from the entry queue. If the aborter does not succeed in removing a task from an entry queue, the aborter will wait until the entry call attempt is over before continuing its execution.

Since only one of the actors will succeed in either resetting the *rdv* flag or removing the entry node there is no race condition. Whatever actor is successful will unblock the task.

Callers and Owners

Protecting each entry queue with an A/B-lock ensures that a dequeue from an entry queue by its owner and an interior removal of the queue by a time out or an aborter cannot happen concurrently. Similarly, they guarantee that an owner cannot concurrently dequeue from an entry queue while a caller enqueues to the queue. Consider, also, the following observations about the owner and caller of an entry.

(*O1*) The *rdv* flag will be obtained by only one of the following actors: an aborter, the owner, the caller, a master or a timer handler. Whichever actor obtains the *rdv* flag has committed the owner to engage in an event with that actor

(*O2*) No callers can be removed from an entry queue while the owner has a A-lock

(*O3*) The owner of an entry is the only task to set the *guard* of the entry The *guard* is set before the *rdv* flag is set.

(*O4*) No inserts to an entry queue may be made while the owner is resetting the *guard* (The owner maintains a A lock)

(*C1*) The caller will be removed from an entry queue by only one of the following actors: an aborter, the owner of the entry queue, or a timer. Whichever actor removes the caller has committed the caller to engage in an event with that actor.

(*C2*) No attempt will be made to remove the caller until after it has been enqueued. (The status of the caller will not be changed until after it has been enqueued.)

(*C3*) The *guard* of an entry cannot be changed while the caller has a B-lock for that entry.

(*C4*) The caller does not increment *entry.count* until after it has successfully reset the *rdv* flag.

Caller vs Owner

From (*O2*) we can deduce that once the owner has decremented *entry.count* and reset the *rdv* flag the owner is committed to rendezvous with the first task on the entry queue. From (*O3*), (*O4*), (*C2*), and (*C3*) we can deduce that once the caller has incremented *entry.count* (and found it to be 0) and reset the *rdv* flag the owner is committed to rendezvous with the caller. (Note the *entry.guard* cannot be reset while the caller maintains a B-lock, thus the *rdv* flag cannot be reset and then set while the caller maintains a B-lock.)

Caller vs Caller

From (*C2*) and (*C4*) we can deduce that is only the first caller of an entry that will attempt to reset the *rdv* flag of the owner. Further, only one of the first caller of the entries will succeed in disabling the *rdv* flag of the owner.

4.7. Abort Management

A task that aborts another task causes that task to become abnormal. An abnormal task is not completed, but is unable to interact with other tasks. All the dependents of an abnormal task also become abnormal.

If an abnormal task is blocked at an accept, an entry call, or a delay statement, then the task is unblocked and completed. Otherwise, the abnormal task must be completed when the task reaches a synchronization point that is one of the following: ARM 9.10 (6) *the end of its activation; a point where it causes the activation of another task; an entry call; the start or the end of of an accept statement;a select statement; a delay statement; an exception handler; or an abort statement.*

The exception TASKING__ERROR must be raised in any task which is either engaged in a rendezvous with or waiting on an entry queue of an abnormal task.

The ARM states that aborts are to be used only in severe situations (Ada has a built-in exception handling mechanism for dealing with expected exceptional situations). Thus, a not terribly efficient implementation of the abort statement may be tolerable (and even preferable), as long as this implementation does not degrade the performance of other RTS functions. Accordingly, no attempt was made to parallelize the implementation of aborts, and hence, they are the only SMARTS function that take time proportional to the number of tasks involved.

4.7.1 Data Structures for Aborts

The *abnormal* flag of the *TCB* is used to implement aborts. The *abnormal* flag of a task is set by the aborter of the task. All tasks check their abnormal flag at abort synchronization points. The *status* of a task is inspected by the aborter of the task to determine what actions are appropriate, i.e., remove the task from an entry queue. The *status* of a task is set by the task itself before and after synchronization points. Since a task that is blocked when it is aborted, must be unblocked, the *num__event* and *event* fields of the *TCB* are also involved in the implementation of aborts.

Before moving on to the actual description of the code,we demonstrate how *abnormal* flags are used to implement aborts.

4.7.1.1 Abnormal Flags

In order to avoid asynchronously wresting control away from a task, in SMARTS a task voluntarily aborts itself after it discovers it has been aborted. To implement this cooperative aborting of tasks, the *abnormal* flag is used.

When a task is aborted its *abnormal* flag is set. All tasks check their *abnormal* flags at synchronization points. If the aborted task is not blocked awaiting a rendezvous or a delay, then the task will discover it has been aborted when it reaches an abort synchronization point. If the aborted task is awaiting a rendezvous or a delay, then the aborter will try to disable the rendezvous or delay. If the aborter succeeds, is it its responsibility to unblock the aborted task with an *abort* event.

The *status* filed of the *TCB* of the aborted task is inspected by the aborter to determine what the current activity of the task is, i.e., executing an entry call. To avoid race conditions the order in which the *status* and *abnormal* flag of a task are set is important. All task check their abnormal flags before and after setting their status (and attempting to engage their current activity). The code that a task *t* executing a simple select and the task aborting *t* must execute follows.

Owner:
```
    if T.ABNORMAL then -- Check abnormal flag.
        abortme(T);
    end if;
    fetch_and_store(T.RDV)  -- Enable rendezvous.
    -- Attempt to rendezvous.
    -- Attempt to rendezvous.
    if can rendezvous then
        T.STATUS := ACTIVE;
        ...
    else
        T.STATUS := SELECT; -- Aborts can now succeed.
        if T.ABNORMAL
        and then fetch&store(T.RDV, 0) = 1 then
        -- If we can disable rendezvous, then we don't need to block.
            abortme(T);
        else
            block_task(T);
        end if;
    end if;
```

Aborter:

```
    if fetch&add(T.ABNORMAL, 1) /= 0 then -- Some else has already aborted T.
        return;
    end if;
    if T.STATUS = SELECT then -- Waiting on a simple select.
        if fetch&store(T.RDV, 0) = 1 then -- We disabled the rendezvous flag.
            unblock_task(T, ABORT);
        end if;
    end if;
```

The code for a task executing a delay is similar to the code given above. The code for a task executing an entry call (which we describe below), however, is more complicated. Note that in the above code, the *status* of *t* is set by *t* after attempting to engage in a rendezvous - an aborter will not even attempt to reset the *rdv* flag of *t* until after its *status* is set. Indeed, it is possible that the aborter of *t* will call an entry of *t* after having set the *abnormal* flag of *t*. This is not problematic as an entry caller must inspect the *abnormal* flag of *t* prior to attempting a rendezvous. Such is not the case for accept statements. That is, a task does not inspect the *abnormal* flag of a caller before accepting its entry call. (An implementation where a task must ensure that its entry callers are not aborted prior to accepting their calls requires excessive locking, and hence, was rejected.) In order to adhere to the ARM, we must ensure that after the aborter *a* of a task *t* finishes its portion of the abort, *a* does not accept an entry call that *t* was in the process of issuing when *t* was aborted. (Note that *t* will inspect its *abnormal* flag before initiate any new entry calls.)

A solution which only slightly increases the entry call code is to have the aborter wait until the call attempt is finished (either because the task discovered it was aborted or initiated in a rendezvous before completing its portion of the abort). Thus, the aborter will not accept an entry call from *t*. Further, this solution does not decrease the parallelism and places the onus upon the aborter. The caller must set its *status* both priori to and after attempting a rendezvous (to let an aborter know its current activity). The code for inspecting *abnormal* flags during the execution of entry call is given below.

Caller:

```
T.STATUS := CALL; -- Aborts can now be attempted.

if T.ABNORMAL then -- Check abnormal flag.
    T.STATUS := COMPLETE;
    abortme(T);
end if;
-- Attempt to rendezvous.
if can rendezvous then
    T.STATUS := ACTIVE;
        ...
else
    entry_enqueue(T, E); -- Aborts can now succeed.
    if T.ABNORMAL then
    -- Aborted
        if entry_remove(T) then
        -- If we can disable rendezvous, then we don't need to block.
            T.STATUS := COMPLETE;
            abortme(T);
        else
            T.STATUS := ACTIVE; -- Either abort or rendezvous in progress.
            block_task(T);
        end if;
    end if;
end if;
```

Aborter:

```
if fetch&add(T.ABNORMAL, 1) /= 0 then -- Some else has already aborted T.
    return;
end if;
if T.STATUS = CALL then -- Waiting on a simple select.
    if entry_remove(T) then -- We disabled the rendezvous flag.
        unblock_task(T, ABORT);
    else
        while T.STATUS = CALL loop
        end loop;
    end if;
end if.
```

Two final comments about our implementation using *abnormal* flags should be made. First, in the presence of pre-emptive scheduling or interrupt entries interrupts must be disabled during the entry call code since the aborter busy-waits. However, interrupts will be disabled during the entry call code already because the entry queue operations entail busy-waiting. Second, all of the statements where the *abnormal* flag of tasks are inspected can be removed without affect the integrity of the rest of SMARTS: if a program does not use abort statements, then these code sequences can be omitted.

4.7.2 Actions of the Aborter

When a task *t1* aborts a task *t2*, *t1* executes a fetch&add(*t2.abnormal*, 1). If the value returned from the fetch&add is 0, i.e., *t1* is the first task to abort *t2*, *t1* will complete the abort. First *t1* will abort the direct dependents of *t2*.

What *t1* does next depends on the *status* of *t2*. If *t2* is executing an accept, *t1* will try to disable the *rdv* flag of *t2*. If *t2* is executing an entry call, *t1* will try to remove *t2* from the entry queue. If the *t2* is executing a delay statement, *t1* will try to disable the time out. If *t1* is successful at disabling the rendezvous, removing *t2* from an entry queue, or disabling the time out, then *t1* will unblock *t2* with an abort event. If *t1* is not successful at removing *t2* from an entry queue, then *t1* must wait until the entry call attempt is over, i.e., until *t2* resets its *status*.

The aborter of a task does not change the *status* of the aborted task, any task that attempts to communicate with the aborted task will first check the *abnormal* flag of the aborted task.

4.7.2 Actions of an Abnormal Task

When a task *t* that was waiting at timed entry call or selective wait is unblocked with an abort event, *t* will attempt to cancel the time out. Note that, the time out has already been effectively disabled: a time out will not succeed unless the SMARTS processor can disable the rendezvous or remove the task from the entry queue (§4.5).

If an abnormal task *t* is not unblocked with an abort event, i.e., either the rendezvous or delay could not be canceled or *t* was was not waiting for a rendezvous or a delay, then *t* will discover that it has been aborted when *t* reaches an abort synchronization point.

When a task *t* discovers that its has been aborted, either because it was unblocked with an abort event or finds that its *abnormal* flag has been set, *t* must unblock all of the tasks that are are currently engaged in rendezvous with *t* or waiting on the entry queues of *t*. The *exception* field of the *TCB* of these tasks is set to *tasking__error* before they are unblocked. Thus, the tasks will raise TASKING__ERROR after they have been unblocked.

t is now completed, so it completes all of its nested blocks. Completing a block amounts to waiting for the dependents of the block to become quiescent (which they will become no later then when they discover that they have been aborted) and then releasing the *TCB*'s of the dependents (after they have released the TCB's of their dependents). After *t* completes its outermost block, if *t* was not waiting on a terminate alternative, then *t* checks its masters for quiescence. Finally, *t* terminates.

The active direct dependents of an aborted task *t* will either be unblocked with an *abort* event by the aborter of *t*, or they will discover that they have been aborted when they reach an abort synchronization point. The direct dependents of *t* that are completed will be unblocked by their direct dependents (after the direct dependents have completed).

4.7.4 Code for Abort Management

```
abort(TASK1; TASK2) is -- Task1 aborts task2.
begin
    abort_task(TASK2);
    if read(TASK1.ABNORMAL) then -- This is a synchronization point.
        abortme(TASK1); --The killer may itself have been aborted.
    end if;
end abort;

procedure abort_task(TASK) is
begin
    if fetch_and_add(TASK.ABNORMAL, 1) /= 0 or read(TASK2.STATUS) = TERMINATED then
    -- Make sure only one task tries to abort task2.
        add(TASK.ABNORMAL, -1); -- Avoid overflow.
        return;
    end if;
    BLOCK := read(TASK.BLOCK_PTR);
    while BLOCK /= null loop -- Kill dependents by block.
        abort_tasks(BLOCK.FIRST_DEPS);
        BLOCK := read(BLOCK.PREVIOUS_BFP);
    end loop;
    -- Try to disable task2 if it is waiting, if successful we must signal, otherwise
    -- task2 will discover task2 is aborted when task2 awakens.
    case read(TASK.STATUS) is
    when TIMED_SELECT =>
        if fetch_and_store(TASK, RDV, false) = true  then -- Disabled rendezvous.
            timed_unblock(TASK, ABORT);
        end if;
    when SELECT, TERM_SELECT =>
        if fetch_and_add(TASK.RDV, false) = true then -- Disabled rendezvous.
            unblock(TASK, ABORT);
        end if;
    when TIMED_CALL =>
        if entry_remove(TASK) then -- Take off entry queue.
            timed_unblock(TASK, ABORT);
        else
            while read(TASK.STATUS) = TIMED_CALL loop
            -- Wait until entry call attempt is over.
            end loop;
        end  if;
    when CALL =>
        if entry_remove(TASK) then -- Take off entry queue.
            unblock(TASK, ABORT);
        else
            while read(TASK.STATUS) = CALL loop
            -- Wait until entry call attempt is over.
            end loop;
        end if;
    when WAIT =>
        if time_remove(TASK2.TIME_NODE_PTR) then
            timed_unblock(TASK, ABORT);
        end if;
    end case;
    -- There is no way another task can tell whether task2 is completed or
    -- abnormal. It will finish the job itself regardless.
    -- Note: t'callable checks t.abnormal.
end abort_task;

procedure abort_tasks(TASK_LIST) is
-- Abort all of the tasks in TASK_LIST (linked by SIBLING pointers).
begin
    while read(TASK_LIST) /= null loop
        if read(TASK_LIST.STATUS) = CREATE or read(TASK_LIST.STATUS) = ACTIVATE
        or read(TASK.STATUS) = TERMINATE then
        -- I.e., if a multi-item.
            for I in 1..read(TASK_LIST.MULT) loop
                abort_task(TASK_LIST.TCB_PTRS(I));
            end loop;
        else
            abort_task(TASK_LIST);
        end if;;
        TASK_LIST := read(TASK_LIST.SIBLING);
    end loop;
end abort_tasks;

procedure abortme(TASK; TERM_FLAG) is
-- Task has discovered it was aborted.
begin
    -- Terminate those task that TASK has created, but not yet activated.
    terminate_unactivated(TASK);

    -- Raise TASKING_ERROR in the tasks that are waiting on my entry queues.
    purge_rdv(TASK);
    -- Raise TASKING_ERROR in the tasks I am engaged in a rendezvous with.
    SERVICED := read(TASK.SERVICED);
    while SERVICED /= null loop
        write(SERVICED.EXCEPTION, TASKING_ERROR);
        if read(SERVICED.STATUS) = TIMED_CALL then
            timed_unblock(SERVICED, END_RDV);
        else
            unblock(SERVICED, END_RDV);
        end if;
        SERVICED := read(SERVICED.NEXT);
    end loop;
    BLOCK := read(TASK.BLOCK_PTR);
    -- Unwind our nested blocks.
    while read(BLOCK) /= null loop
        if fetch_and_add(BLOCK.NO_TERM, -1) /= 1 then
        -- We have active dependents, so wait for them to be quiescent.
            block_task(TASK);
        end if;
        if fetch_and_add(BLOCK.NUM_DEPS, -1) /= 1 then
         -- Wait for our dependents to free the TCBs of their dependents.
            block_task(TASK);
        end if. - Release the tcbs of my direct dependents.
        uncreate_tasks(BLOCK.FIRST_DEP);
        BLOCK := read(BLOCK.BLOCK_PTR);
    end loop;

    if TERM_FLAG = NOT_TERM and then fetch_and__and_add(TASK.NO_TERM, -1) = 1 then
    -- If I haven't checked for quiescence already.
        check_term(TASK.MASTER_TASK, TASK.MASTER_BLOCK)
    end if;

    -- Terminate the task.
    write(TASK.STATUS, TERMINATED);
    free(TASK.DATA_PTR) -- Free my stack space.
    -- Check if our master task, or master block is completed.
    check_done(TASK.MASTER_TASK, TASK.MASTER_BLOCK);
    initialize_pe; -- Get a new task.
end abortme;
```

4.8. Shared Variables and Stack Management

Resolving the non-local references of tasks is more complicated than resolving the non-local references of subprograms because there is more than one active task and possibly more than one level of memory (e.g., local and global). Common stack management schemes that rely on displays or static links to resolve references to non-local data assume a linear address space and, therefore, must be altered for multiprocessor architectures with a hierarchical memory.

Another method for resolving the non-local references of block structured languages which can be extended to parallel languages and multiprocessors quite naturally is the so called *relay set* [Buroff 1977][15]. Relay sets have been used by Kruchten to resolve the non-local references of tasks for an implementation of Ada on a uniprocessor [Kruchten 1985]. In this section we extend this work: we show how to use relay sets to efficiently resolve the non-local references of Ada tasks in a multiprocessor environment. When applied to a task, a relay set contains all of the non-local variables referenced within the task. These variables are passed to the task as if they were parameters passed by reference.

Relay sets also provide a means for calculating the *shared set* of a program unit. The shared set of a program unit is the set of variables declared by the program unit that are referenced non-locally by tasks. (Not all Ada shared variables must be explicitly declared as shared.) It is imperative that these variables be identified, so that they can be allocated in the proper memory (e.g., global as opposed to local) and accessed efficiently. (Variables that are not in the shared set of any program unit are called *private*.)

In this section, we present two schemes, one based on relay sets and the other based on shared sets, for efficiently managing Ada variables on a multiprocessor with a three tier memory hierarchy (caches, local memories and a global memory).

[15] Buroff used relay sets, or *total displays* as he called them, in an implementation of ALGOL 68 to eliminate scope and name restrictions.

We begin by reviewing the Ada rules for shared variables. Next, we consider the appropriate mapping of Ada variables onto the storage modes (defined in §2.3.1) of our machine model. Some of the problems with using traditional stack management techniques to implement Ada variables with these storage modes are then identified. After defining relay sets and shared sets, we explain how they can be used to efficiently resolve the non-local references of tasks in a multiprocessor environment with a hierarchical memory. In §4.8.4 and §4.8.5, we show in detail how to compute the relay sets and shared sets of program units. An example of the calculation of the relay and shared sets is given in §4.8.6. Finally, we assess the cost of using relay sets and shared sets to manage Ada variables.

4.8.1 Ada Shared Variables

A task may reference a variable declared in an enclosing task. Such a variable is said to be shared. The Ada reference manual ARM 9.11 (7-8) states that an implementation of Ada is allowed to maintain local copies of shared variables, and only need update the variables at points where tasks synchronize. A program is erroneous if keeping local copies of a shared variables produces different results than not keeping local copies, i.e., it is up to the programmer to guarantee that the local copies are consistent between task synchronization points.

The synchronization points of tasks that share a variable are defined in the ARM 9.11 (2) to be "*Two tasks are synchronized at the start and end of their rendezvous. At the start and end of its activation, a task is synchronized with the task that causes this activation. A task that has completed its execution is synchronized with any other task.*"

For variables named in a pragma SHARED, every read or write is a synchronization point for that variable. As we defined the terms synchronous and asynchronous variables in §2.3.1, shared variables that are not named in a pragma SHARED are synchronous variables and variables that are named in a pragma SHARED are asynchronous variables.

The pragma SHARED can only be applied to variables of scalar or access type. It cannot be applied to not composite objects. To allow tasks to access the (scalar or access type) components of composite shared objects asynchronously, i.e., to access them as if they were declared in a pragma SHARED, SMARTS provides the pragma VOLATILE. Every read or write of the scalar or access type components of a composite object named in pragma VOLATILE is a synchronization point for that component.

Implicitly Shared Variables and Separate Compilation

Although in direct violation of Steelman (the design requirements document of Ada [DoD 1979]), one consequence of the Ada rules for shared variables is that there is no way for a task to explicitly declare its intention of sharing a variable synchronously. Synchronous variables become shared implicitly by virtue of being accessed by a task other than the task that declares the variables.

```
procedure outer is -- Example of inner packages and outer units sharing variables.
    task type share_i;
    ti : share_i;
    j : integer;
    package inner is
        i : integer;
    end inner;
    package body inner is separate;
    -- Note that inner has not been compiled, thus its shared set is not known.
    task body share_i is
    begin
        inner.i := 0; -- shared.i is shared as it is referenced in share_i.
    end share_i;
begin
    null.
end outer;
------------------- ------ ------------- --- -------------
separate(outer);
package body inner is
-- Note that outer has been compiled, but its set of directly declared task types
-- is incomplete.
    i : integer;
    begin
    declare
        task type share_j;    Directly declared task type of outer
        tj   share j;
        task body share_j is
```

```
          begin
              j := 0; -- outer.j is shared as it is referenced in share_j.
          end share_j;
      begin
          null;
      end;
  end inner;
```

Not requiring synchronous shared variables to be identified as shared when they are declared has some severe repercussions. In the presence of separate compilation, a compiler can not be sure what variables are shared until bind time (since variables declared in one compilation unit could be referenced by tasks declared in a parent compilation unit). Either code generation must delayed until that time, or pessimistic assumptions must be made, e.g., all the variables that appear before the declaration of a subunit that is compiled separately must be regarded as shared. Furthermore, some otherwise safe optimizations cannot be performed when shared variables are involved, these too may have to be delayed until bind time [AI-00315].

4.8.2 Storage Modes of Ada Variables

Recall from §2.3.1 that the virtual address space of our target machine is composed of segments that have storage modes corresponding to the six data storage modes that are the cross product of the software specifiable data attributes: (*local, global*) × (*cacheable, marked, noncacheable*). The data of a each storage mode must be allocated in a segment of the same storage mode

The appropriate storage modes for Ada variables are:

a) Asynchronous Variables:	<*global, noncacheable*>
b) Synchronous Variables:	<*global, marked*>
c) Private Variables	
i) Migratory Tasks	<*local, cacheable*>

ii) Non-migratory Tasks: *<global, cacheable>*

Thus, there are at least three different types of virtual address segments that should be managed by SMARTS. (Later, we see that the cost of stack management schemes is proportional to the number of storage modes supported.)

It is conceivable that an optimizing Ada compiler might want to support storage modes which do not correspond directly to Ada variables. For instance, if a task references enough synchronous shared variables to fill an entire cache, then the compiler might decide to declare the private variables of the task as *<local, noncacheable>*, since caching private data might evict shared data from the cache. That is, the savings from caching a private datum, may be greater than the cost of caching a private datum: a potential cache rather than a local memory reference to the private datum vs a potential global rather than a cache reference to the shared datum that is evicted from the cache.

The advantage of declaring synchronous shared variables with the attribute *marked* is that we can selectively flush these variables from the cache at shared variable synchronization points. As a store-through caching policy is used in our machine model for data declared as *marked*, the *marked* data in the cache need only be invalidated when it is flushed If tasks are migratory, then their private variables must be flushed from the cache at block points, i e., at points where the tasks block Since a store-in caching policy is used for *cacheable* data, the *cacheable* data in the cache must actually be evicted when it is flushed

The points where a task must block vs the points where shared variables must be flushed from the cache are listed in table 4.8.1. As can be seen from the table, block points and shared variable synchronization points are not identical For the migratory task case, if block points and shared variable synchronization points coincided, then it would not be necessary to selectively flush synchronous data from the cache at shared variable synchronization points (since the entire cache would be flushed) However, even in this case, it is still advantageous to differentiate between synchronous and private variables because they have different access

Event	Block Point	Shared Variable Synchronization Point
Parent starts activation of child.	yes	yes
Child completes activation.	no	yes
Caller calls entry.	yes	maybe
Owner reaches an accept statement.	maybe	no
Owner starts rendezvous.	no	yes
Owner completes rendezvous.	no	yes
Task delays.	yes	no
Task completes (or terminates).	yes	yes

Table 4.8.1: Block Points vs Shared Variable Synchronization Points.

patterns. The results of simulations by McAuliffe [McAuliffe 1986] suggest that, due to their low frequency of stores, synchronous data should be cached using a store-through policy, whereas private data should be cached using a store-in caching policy. By declaring synchronous variables as *marked*, we ensure that a store-through caching policy is used for these variables.

4.8.3 Storage Modes and Stacks: The Data Placement Problem

In this subsection, we consider some of the difficulties of managing variables with different storage modes. In general, the complexity of the schemes for resolving non-local references of tasks depends on the number of storage modes supported. As discussed in §4.8.9, there are, however, clear time/space tradeoffs between the various schemes.

Block structured languages that support recursive subprograms are usually implemented with an activation stack. When a subprogram is called, an activation record that contains the local data of the subprogram is pushed onto the stack. Non-local references are resolved using static links or a display. With static links each activation record is linked to the activation record of its (statically) enclosing scope. Displays collect all the pointers to the activation records of (statically) enclosing scopes in a table.

For languages with parallel constructs, such as Ada, the activation stack of a program becomes a *cactus* stack. The branches of the cactus stack are the activation stacks of tasks. A snapshot of the activation stack that could result from the execution of the program given in §4.8.8 appears in the diagram below.

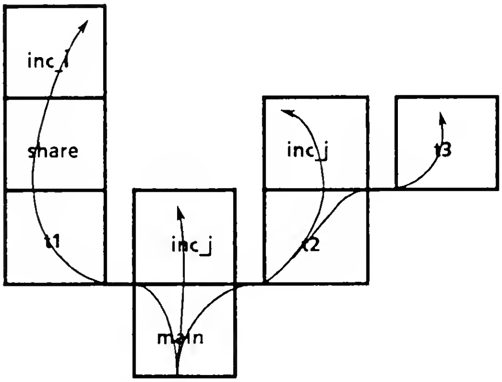


Figure 4.8.1. Displays: A Cactus Stack

Since there may be several levels of indirection, dereferences via static links to activation records that reside in global memory or in the local memory of a different processor may be expensive. While there is only one level of indirection when displays are used, maintaining a display for a cactus stack is also problematic: there may be more than one display pointer per nesting level (one for each active task at the same level). Thus, a single global display (which is the typically implementation of displays for languages without parallel constructs) cannot be used for languages with parallel constructs. Instead, each task must keep a local copy of its display (see for example, [Lampson 1982]) which is more expensive in terms of both time and storage than keeping a global display.

A more serious problem with cactus stacks (as they have traditionally been implemented) is that there is no clear demarcation between data with different storage modes within an activation record. Consecutive stack locations may contain data with different storage modes. But, data of different storage modes must be allocated in different segments. Furthermore, as a stack grows and shrinks the same stack location may refer to data of different storage modes at different times. Again, data of different storage modes must be allocated in different segments. We will call this the data placement problem.

One possible solution to the data placement problem is to maintain a separate cactus stack, and hence display, for each storage mode. Assuming we support *m* storage modes, this would mean keeping *m* local displays and stack pointers per task which is quite expensive in terms of storage and registers. (For comparison purposes, we assume in this discussion that stack and display pointers are kept in registers whenever possible.)

Another approach, taken by the SYMUNIX operating system, is to use *virtual aliases* [Lipkis *et al.* 1987]. Every stack address is mapped to *m* virtual addresses each with a different storage mode. The upper order bits of a stack address determine which of the virtual addresses it is currently mapped to. These bits are set according to the storage mode of the data being stored at the stack address. Unfortunately, this scheme effectively divides the virtual address space by *m* (without reducing the number of bits required to represent an address). Moreover, as most operating systems will not provide this novel address translations scheme, assuming its existence would severely limit the portability of SMARTS.

Stack management for Ada variables with different storage modes is further complicated by the presence of implicitly shared variables and separate compilation. We do not know whether a variable declared in one task (that is not named in a pragma SHARED) is shared until it is referenced by another task. Hence, when a subunit is compiled separately, we cannot be certain of the storage modes of the variables declared before the subunit until bind time. Thus, it is not until bind time that we know where to allocate these variables.

4.8.4 Overview of Relay Sets and Shared Sets

An alternative to static links and displays for resolving the non-local references of block structured languages are relay sets [Burhoff 1977]. The relay set of a subprogram identifies which non-local variables the subprogram references. These variables are passed to the subprogram as if they were parameters passed by reference.

Like displays only one level of indirection is needed when relay sets are used. Unlike displays, the storage requirements of relay sets depends on the number of non-local references, not on the static nesting level. Using relay sets for block structured languages leads to a *rake* of scopes in which a shared heap is the cross bar and the stacks of the subprograms are the prongs. All referenced non-local data are kept in the heap.

4.8.4.1 Relay Tables

For a parallel block structured language the relay set can be modified so that there are multiple heaps (one per storage mode), and each prong is the stack of one of the tasks. (The main procedure is regarded as a task.) A snapshot of the rake of stacks that could result from the execution of the program given in §4.8.8 appears in figure 4.8.2

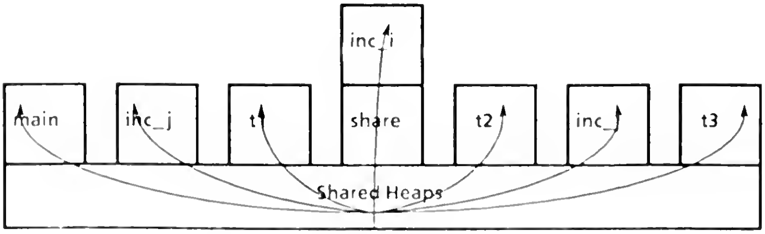


Figure 4.8.2 Relay Tables: A Rake of Stacks

The first activation record of a task contains its *relay table*, where the addresses of its relay set will be stored. Since the relay set of a task captures its environment, it is possible to use relay sets for tasks, while using displays within tasks for subprograms. (There are some slight complications with this scheme. These are addressed in §4.8.6.3). We will call this approach, the *relay table* scheme.

The shared set of a program unit can be calculated from the relay sets of its inner tasks (see §4.8.7 for details on the calculation of shared sets). The variables in the shared set of a program unit are allocated in the heap of the appropriate storage mode. Only pointers to these variables are kept in the program unit's activation record. Thus, relay sets and shared sets elegantly solve the data placement problem.

Since pointers are being kept, two levels of indirection are required to access a non-local shared variable (one via the stack pointer and another via the shared table entry), whereas, only one level of indirection is needed for multiple cactus stacks. However, as the addresses of the shared variables are read only (after being written by the declarer of the shared variables), local copies of the addresses can be cached or kept in registers. Hence, the penalty of the extra level of indirection need only be paid the first time a shared variable is referenced (at the expense of of some extra storage).

4.8.2.2 Shared Tables

Shared sets can be used in conjunction with displays to efficiently solve the data placement problem for cactus stacks: Each task maintains two stacks, one for private data and the other for shared data. The shared stack is a cactus stack that consists of *shared tables*. The shared table of a program unit contains an entry for each element of its shared set. Rather than allocating the elements of its shared set in an activation record, a program unit allocates these elements in a heap of the appropriate storage mode, and stores pointers to these heap objects in its shared table.

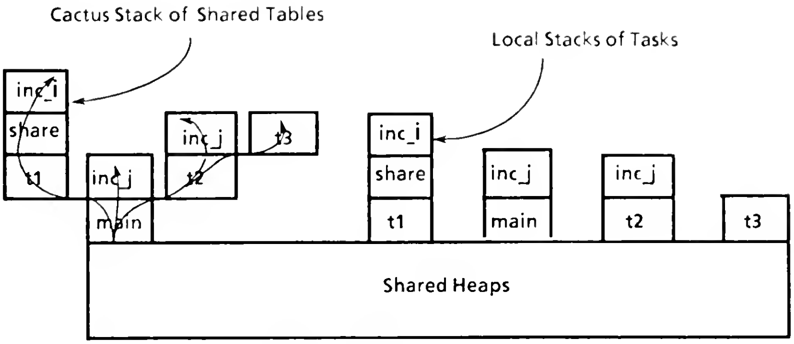


Figure 4.8.3. Shared Table: Cactus Stack and Rake of Stacks

If task are migratory, then the private and shared stacks can be coalesced since they both would be allocated in the global memory. In this case, the shared tables would be stored in activation records.

As with relay tables, two levels of indirection are required to access a non-local shared variable (one via the display pointer and another via the shared table entry). The relative costs of using multiple stacks, relay tables and shared tables are discussed in detail in §4.8 9 To implement both the relay table scheme and shared table scheme, we must be able to identify the shared sets of program units. Fortunately, as described in the following sections, relay sets provide us with such a method.

4.8.2.3. Separate Compilation

The algorithms given in the next two subsections for calculating relay and shared sets assume that all separately-compiled subunits of a unit being processed have been compiled When subunits of unit are compiled separately, then the actual calculation of relay set of the unit must be delayed until bind time, however, code for the unit can still be generated at compile time. Unfortunately, to be able to generate code at compile time for a unit whose subunits have not been compiled, we must be overly pessimistic about the contents of its shared set.

It is not obvious that code can be generated at compile-time for units with separately compiled subunits when relay sets are used to resolve non-local references: Since a subunit may reference variables declared in outer units, the size of the relay table of a unit may grow after a subunit has been compiled. An important observation made by Kruchten [Kruchten

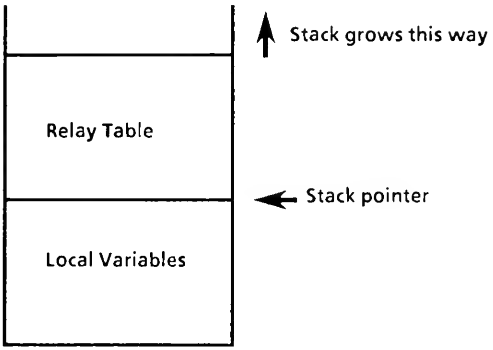


Figure 4.8.4. Activation Record Containing Relay Table

1985] is that the values of a relay table are not needed to generate code for a unit, only the offset from the stack pointer of those elements of the relay table that are actually referenced in the unit. By placing the relay table above the other local variables in the stack frame, we ensure that the offsets of local variables will not be affected by the size of the relay table. Thus, while it may sometimes be necessary to delay the calculations of some relay sets sets until bind time, code can be generated when a program unit is compiled

The storage mode of a local variable will determine whether it is allocated in an activation record or in one of the heaps, and hence, how it must be accessed. Thus, we must know which of the local variables of a program unit are shared in order to generate code for the unit. The shared set of a program unit depends on precisely which of its variables are referenced by its inner subunits. In the presence of separately-compiled subunits, the feasibility of calculating the actual shared set of a unit at compile-time is bleak, but not completely hopeless: the shared set of a program unit can still be calculated at compile-time when its subunits do not contain tasks.

To allow efficient code to be generated at compile-time for units whose subunits do not contain tasks, we provide a pragma, NO_TASKS, that is used to inform SMARTS that a separately-compiled subunit does not contain any tasks. A pragma NO_TASKS that names a subunit must appear in the same scope as the subunit specification. If a subunit of a program unit is not named in a pragma NO_TASKS, then we must assume that all variables declared before the subunit are shared, or else delay code generation until bind time (when the shared set of the unit can be accurately calculated).

While the shared sets of separately-compiled subunits that are task or subprogram bodies can be computed accurately at compile time (assuming that they do not themselves have subunits that declare tasks), the same is not true of subunits that are package bodies. The parent unit may contain other subunits that declare tasks, and these subunits have direct visibility of the package specification.

In general, all the variables in the visible part of a library package must be considered as shared, since there is no way of knowing how these variables will be accessed. (However, this is not problematic as the data belonging to library packages is in the global scope, and therefore, need not be in any relay table – see the next section.) As a canonical example of the sharing of library packages, consider the predefined inout-output package TEXT_IO. The default standard input and output files, STANDARD_INPUT and STANDARD_OUTPUT, will be accessed by any task that reads from standard input or writes to standard output. Hence, we must assume that STANDARD_INPUT and STANDARD_OUTPUT are shared.

4.8.5 The Relay Set

In this subsection, we present the algorithm given in [Kruchten 1985] for calculating the relay sets of Ada program units. After doing so, we examine more closely the mechanics of using relay sets for resolving the non-local references of subprograms and tasks.

In block structured languages, there are three kinds of objects visible to a program unit:

- Global Entities - Entities local only to the outermost scope, i.e., library packages.
- Local Entities - Entities defined immediately within the declarative part of the scope, not within an inner scope.
- Semi-global Entities - Entities local only to an outer scope other than the outermost scope, i.e., all entities which are neither global nor local.

We use the term *entities* in its most general sense to mean objects, types, parameters, subprograms, entries and tasks.

The set of semi-global entities that a unit references is called its *relay set*. When using a relay set, all the semi-global entities that are referenced by the program unit are treated as parameters that are passed to the program unit by reference. It is important to note that the relay set of a unit includes not only those entities that it references directly, but also those entities that any subunit within it references. Program units whose relay set are empty can be treated as global entities, as their environment is static.

Formally, the relay set RS_p of semi-global entities referenced by a program unit p is defined as

$$RS_p = \left(\bigcup_{q \in L_p} RS_q \right) \cup R_p - L_p - G,$$

where q is a subprogram, L_p is the set of entities local only to p, R_p is the set of entities referenced by p, and G is the set of global or static entities.

4.8.5.1 Computing the Relay Set of a Program Unit

From the definition, we get the following algorithm to compute the relay sets of program units. For the purpose of this calculation, a block statement may be regarded as a call of an anonymous parameterless procedure with a corresponding body declared in the innermost enclosing declarative region. The algorithm takes place over a single pass of the compiler. Initially $G := \{\}$, and each program unit p is processed as follows:

1) $RS_p := \{\}$.

2) $L_p := \{\}$.

3) When a variable, entry or subprogram e is declared local only to p,

   $L_p := L_p \cup \{ e \}$.

4) When an inner program unit q has been processed, if RS_q is empty, so that q is static,

   $G := G \cup \{ q \}$,

otherwise

   $L_p := L_p \cup \{ q \}$

and

   $RS_p := RS_p \cup (RS_q - L_p)$.

Note Although L_p may still be incomplete because all of the declarations of p have not been seen yet, the visibility rules of Ada ensure that q does not reference any entities declared later in p

5) When a reference is made to an entity e that is not already in G or L_p,

   $RS_p := RS_p \cup \{ e \}$

4.8.5.2 Relay Sets of Subprograms

When the compiler processes a the body of a subprogram, its relay set is calculated as specified above. For each subprogram declaration a subprogram template is built. This template contains all the static structural information of the subprogram, including the local addresses of the relay set.

At run time a subprogram object is created from the subprogram template. Part of a subprogram object is its relay table. The relay table of the subprogram object will contain the run time addresses of the relay set. These actual addresses are calculated when the subprogram body is elaborated. Elaboration consists of:

a) getting the actual addresses of the relay set from the current stack frame (the actual addresses will either be the addresses of objects local only to the current scope or the addresses in the relay table of the current scope) and

b) storing these addresses in the locations of the relay table of the subprogram or object as specified by the subprogram template.

Once a subprogram has been elaborated, it can be called. When a subprogram is called, its relay table is copied into the new stack frame. Note that a subprogram object is not created for each call, but only once upon elaboration. However, a subprogram q within a subprogram p must be elaborated anew for each call of p.

4.8.5.3 Relay Sets of Tasks

The creation of task templates and objects is very similar to the creation of subprogram templates and objects. A task can be activated after the corresponding task body is elaborated. Note that the relay table of a task object is not calculated anew for each task designated by objects of the task type, but only when the corresponding task body is elaborated. (For example, when an array of tasks is created, their task body is elaborated only once.)

Relay tables and displays are for the most part orthogonal, in that relays tables can be used for tasks while displays are used for other program units[16]. (Note, however, that we must still calculate the relay sets of other program units to calculate the relay sets of tasks.) However, relay tables and displays are not completely separable as tasks can call subprograms that are local to other tasks: the non-local variables referenced by a shared subprogram in general will not be in the relay set of (or local to) a calling task.

```
procedure main is  -- Example of how two tasks can share a subprogram.
    i :  integer : = 0;
    task type tt;
    t1, t2 :  tt;
    procedure share is
    begin
        i : = i + 1;
    end;
    task body tt is
    begin  -- Since the procedure share is shared so is i.
        share;
    end;
begin
    null;
end;
```

To resolve the non-local references of shared subprograms, either shared subprograms must have a relay tables, or the relay set of shared subprogram must be added to the relay set of any task that calls the subprogram. We prefer the later scheme, as it allows shared and non-shared subprograms to be handled in a uniform manner at run-time Furthermore, a task can always distinguish between local and non-local subprogram calls, and hence, know when to add the called subprogram's relay set to its own Adding the relay set of shared subprogram to the relay sets of any tasks that call the subprogram is also attractive when

[16] A subprogram that is designated as the main program must be treated as a task However, a main program does not need a relay table as the only non local references the main program can make are to static variables

relay sets are not used to resolve non-local references, as it simplifies the detection of the shared sets of program units (see §4.8.6).

To reflect this change, we formally define the modified relay set MRS_p of semi-global entities referenced and the relay sets of shared subprograms invoked (directly or indirectly) by a program unit p as

$$MRS_p = \left(\bigcup_{q \in L_p} MRS_q \right) \cup R_p - L_p - G + \left(\bigcup_{s \in CS_p} MRS_s \right),$$

where q is a program unit, L_p is the set of entities local only to p, R_p is the set of entities referenced by p, G is the set of global or static entities, s is a subprogram, and CS_p is the set of non-local subprograms called by of p.

There are two points that should be made regarding the definition of modified relay sets. First, we must calculate the modified relay sets of subprograms even though their non-local references are resolved using displays. A shared subprogram s that is called by a task t may invoke other subprograms; the relay sets of these indirectly shared subprograms must be included in the relay set t. Second, adding the modified relay sets of its inner subprograms (rather than the non-modified relays sets) to the relay set of a task, will not increase the size of the relay set of the task: The relay sets of any subprogram local to the task called by its inner subprograms will already be contained in the relay set of the task. (It is precisely the relay sets of any non-local subprogram called from within the task that we wish to add to the relay set of the task.)

As with synchronous shared variables, a subprogram declared by one task becomes shared simply by virtue of being accessed by another task. In the presence of separate compilation, we may be unable to determine precisely what subprograms contained in a unit are shared until bind time. In a similar vein, the relay set of a subprogram contained in a separately-compiled subunit may not be available when a task that calls the subprogram is compiled. Thus, the binder will have to add the relay sets of separately-compiled shared subprograms to the relay sets of any task that call the subprogram.

Even when relay tables are not used for subprograms, it may be desirable to associate relay tables with accept statements to facilitate optimizations wherein accept statements can be executed by the calling or called task[17]. The primary difficulty in implementing such optimizations efficiently is that the rendezvous must executed in the environment of the called task. An entry is declared in the specification of a task, while the accept statement for the entry appears in the body of the task. Any task that can see the task specification can call the entry. Thus, the environment of the accept statement may include scopes that are not in the local display of the calling task. However, the environment of an accept statement is completely captured in its relay table.

4.8.6 Detecting Shared Variables

The shared set of a task is the set of variables local to the task that are referenced by other tasks. The shared set of program unit that is not a task, i.e., a subprogram or package, is the set of variables local only to the program unit that are referenced by a task other than the immediately enclosing task. Asynchronous shared variables are easily identified as such by the pragma SHARED In this section we show how synchronous shared variables can be identified using relay sets and shared sets.

Because of the visibility rules of Ada, it is only those tasks whose types are declared within a given task or subprogram that can name its entities We will call the tasks whose types are declared within a program unit *p*, but not within any inner task, the directly declared tasks of *p* When a variable that is local to *p*, is referenced by a task that is inner to

[17] Whichever task arrives at the rendezvous first will block The task which arrives second will execute the code associated with the accept statement Thus, only one task will have to block during a rendezvous

one of the directly declared task types of p, then the variable will be in the relay set of that directly declared task. Thus, the shared variables of a task will consist of those variables declared by the task that are contained in the relay sets of its directly declared task types. The shared set of a subprogram is defined analogously. When packages are not compiled separately, they can be promoted to the scope of the enclosing program unit, and pose no special problems for calculating shared sets. (When a package is compiled separately, then all of the variables declared in its visible part may have to be included in its shared set - see 4.8.4.3.)

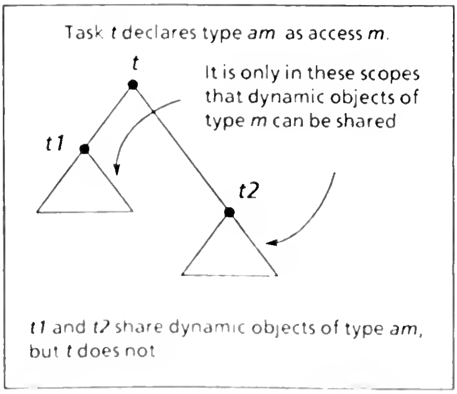
Things are actually a bit more complex because of shared subprograms. The entities declared by a program unit that are referenced by a shared inner subprogram must also be considered as shared. However, if, as suggested in §4.8.6.2, the relay set of a shared subprogram is added to the relay sets of any task that calls the subprograms, then these variables will be in the relay sets of the directly declared tasks types of the task that declares the subprogram. In the algorithm given below, we assume that the relays set of a shared subprogram is added to the relay sets of calling tasks. i.e., we use modified relay sets.

Another complication is that it is difficult to determine which object a shared access object designates: the object designated by an access object may be shared even when the access object is not shared. Arrays pose a problem similar to that of the collection of objects designated by a given access type: it is often difficult to tell which elements of an array are shared. Unless rigorous data flow analysis is performed, the whole collection of objects designated by a given access type and the entire array must be considered shared However, neither of these complications are too serious in practice: When objects are created from access objects at run-time it is known whether such access objects are shared, and hence, the objects can be allocated in the proper type of storage. Array are not typically allocated on the activation stack, so again, the array can be allocated in the proper type of storage art run-time. In the next subsection, we show how to obtain a sharper estimate of what objects designated by a shared access object are actually shared.

4.8.6.1 Shared Dynamic Objects

A problem arises from sharing access objects. While it is not that difficult to detect which access objects are shared, it is harder to tell what an access object designates when it is shared. In general, parameters of accept statements or shared subprograms do not have to be considered as shared; however, when a task passes a locally declared access object to another task which assigns it to an access object local to this called task during the entry or subprogram call, then the designated object can become shared.

The simplest solution to this possible sharing of designated objects is to add all designated objects that have the same type as a shared access object to the set of shared variables of a program unit. Actually we can do this on a task by task basis. A task that does not share any of a given type of access object need not add its local access objects of the same type to its set of shared variables. Consider the diagrammed case.not reference any dynamic objects of type *m*


Task *t* declares type *am* as access *m*.
t
It is only in these scopes that dynamic objects of type *m* can be shared
t1
t2
t1 and *t2* share dynamic objects of type *am*, but *t* does not

Suppose that the task *t* does not share any objects of type *am* (access *m*) with its dependent tasks *t1* or *t2*, while tasks *t1* and *t2* share objects of type *am* (say via parameter passing). Since *t1* and *t2* do not share any objects of type *am* with *t*, *t1* and *t2* do not reference any dynamic objects of type *m* created by *t*. Furthermore, *t* does created by *t1* or *t2*. Therefore its is impossible for the dynamic objects of type *m* that are created by *t* to become shared. Hence, it is only those dynamic objects of type *m* created by *t1* and *t2* that may become shared. The other cases are similar.

4.8.6.2 Calculation of the Shared Set of a Program Unit

The shared set of a program unit can be computed while the modified relay sets of its directly defined task types are being calculated. Formally the shared set SS_p of a program unit (i.e., task , subprogram or block) p is defined as

$$SS_p = \left(\bigcup_{q \in DT_p} MRS_q \cap L_p \right) + \left(\bigcup_{ao \in SS_t} type(o, p) \right)$$

where q is a task, DT_p is the set of directly defined task types of p, MRS_q is the modified relay set of q, L_p is the set of entities (other than constants, types and tasks) declared immediately in p, t is the immediately enclosing task of p (which we define to be p itself when p is a task), ao is an access object and *type(o, p)* is the set of objects declared in p that are designated by access objects of the same type as ao (ao may be shared because it is a parameter or because of use). To calculate the synchronous shared variables of a task we must remove the asynchronous shared variables of the task from the shared set of the task.

Below we give the algorithm to calculate the shared sets of program units. The algorithm can be simplified when the program unit is task, as p will equal t.

Algorithm to calculate the shared set of a program unit p:

1) $SS_p := \{\}$.

2) $L_p := \{\}$.

3) When an entity e other than a constant, type or task is declared,

   $L_p := L_p + \{ e \}$.

4) When a subprogram body s has been processed,

$DT_p := DT_p + DT_s,$

and for each task $q \in DT_s,$

$SS_p := SS_p + (MRS_q \cap L_p).$

5) When a task body q has been processed,

$DT_p := DT_p + \{q\},$

and

$SS_p := SS_p + (RS_q \cap L_p).$

6) When processing a call to an entry or to a subprogram that is local to a parent of t, with a parameter ao that is an access object,

$SS_t := SS_t + \ type(o, t),$

where $type(o, t)$ represents the set of all objects in t that are designated by access objects of the same type as ao.

7) After p has been processed, then for each $ao \in SS_p$:

$SS_t := SS_t + type(o, t).$

8) After t has been processed, then for each $type(o, t) \in SS_t$:

$SS_p := SS_p + type(o, p).$

Note that steps *6)*, *7)* and *8)* could be implemented by simply setting a flag in the type template for ao, i e., they do not require a separate pass

4.8.7 An Example of the Calculation of Relay and Shared Sets

The following program has been annotated with its relay and shared sets

```
procedure main is  -- SSmain = { i, share, ao1, ao2, ao3, ao4, ao5  inc_j, j }
    type at is access integer;     -- MRSmain = { }
    i, j    integer  := 0;
    pragma shared(j);
    ao1      at
    ao2, ao3     at    := new integer;
    task t1;
    task  t2;
    procedure share(ao4 :  in at, ao5 :  out at) is
                                    -- MRSshare = { i , ao1, ao2 }
        procedure inc_i is          -- MRSinc_i = { i }
        begin    -- inc_i
            i : = i + 1;
        end inc_i;
    begin -- share
        ao1 : = ao4;   -- The object that ao4 designates is available to main.
        ao5 : = ao2;   -- The object that ao2 designates is available through ao5.
        inc;
    end share;
    procedure inc_j is              -- MRSinc_j = { j }
    begin    -- inc_j
        j : = j + 1;
    end inc_j;
    task body t1 is                         -- SSt1 = { ao6, ao7, ao8 }
        ao6, ao7 :  at : = new integer;     -- MRSt1 = { share, i , ao1, ao2 }
        ao8 :  at;
    begin -- t1
        share(ao6,ao8);
    end t1;
    task body t2 is                         -- MRSt2 = { inc_j, j }, SSt2 = {  }
        task t3 is                          -- MRSt3 = { inc_j, j }, SSt3 = {  }
        begin -- t3
            inc_j;
        end t3;
    begin  -- t2
        inc_j;
    end t2;
begin -- main
    inc_j;
end main;
```

The relay sets are calculated as follows. The relay set of *inc_i* and *inc_j* are simply the non-local variables they reference:

$RS_{inc\_i} = \{ i \}, MRS_{inc\_j} = \{ j \}.$

The relay set of *share* includes the relay set of its inner subprogram *inc_i*:

$RS_{share} = \{ i, ao1, ao2 \}.$

The relay set of *t1* must include the relay set for *share* as it is a non-local subprogram

$MRS_{t1} = \{ share, i, ao1, ao2 \}.$

Similarly, the relay set of *t3* must include the relay set for *inc_j*:

$RS_{t3} = \{ inc\_j, j \}$.

The relay set of *t2* must include the relay set of its inner task *t2*.

$RS_{t2} = \{ inc\_j, j \}$.

Finally, the relay set of *main is* empty, as all variables referenced in *main* are contained in its scope:

$RS_{main} = \{ \}$.

The shared sets of *t1* and *main* are calculated as follows:

ao6 and *ao8* parameters of the non-local subprogram *share* $\Rightarrow SS_{t1} \supseteq \{ ao6, ao7, ao8 \}$,

$\{ i, share \} \subset MRS_{t1} \Rightarrow SS_{main} \supseteq \{ i, share \}$,

$\{ ao1, ao2 \} \subset MRS_{t1} \Rightarrow SS_{main} \supseteq \{ ao1, ao2, ao3, ao4, ao5 \}$,

$\{ j, inc\_j \} \subset MRS_{t2} \Rightarrow SS_{main} \supseteq \{ j, inc\_j \}$.

We have,

$SS_{t1} = \{ ao6, ao7, ao8 \}$,

and

$SS_{main} = \{ i, share, ao1, ao2, ao3, ao4, ao5, j, inc\_j \}$.

4.8.8 Cost Considerations

We will compare the costs of using relay sets and shared sets to the costs of using more conventional methods for resolving non-local references. We consider subprograms and tasks separately. In both cases, for comparison purposes, we will assume in our analysis that stack and display pointers are held in registers whenever possible. For subprograms, we compare relay sets to static links and global displays. For tasks, we compare the relay table and shared table schemes (described in §4.8.4) to each other and to the multiple cactus stacks scheme (described in §4.8.3). There are three main costs that we need to consider: compile-time costs, run-time costs, and storage costs. (We include register usage in storage costs.) In addition, there are less quantifiable benefits to using one scheme over another that we will try to asses,

There are tradeoffs between the schemes, and none is categorically better than the others for resolving the non-local references of a given programming language. The most appropriate scheme will depend on what features are supported by the language, and, perhaps more importantly, on what programming idioms are practiced by programmers of the language.

4.8.9.1 Costs for Subprograms

For concreteness, we will assume that displays are implemented using the following scheme which is very efficient in terms of storage and time ([Aho 1986], [Gries 1971], [Harris 1985]). The implementation uses a global display; the display pointer at level *i* is overwritten when a subprogram at level *i* is called; the overwritten display pointer is saved in the activation record of the called subprogram. At subprogram entry or exit only this display pointer must be updated.

Compile-Time Costs

The non-local references of subprograms require no special compile time treatment when static links or displays are used (only the code to resolve the references must be generated. At compile time the relay set and relay table of the subprogram template must be built.

Run-Time Costs

Supplying the addresses for the relay table of the subprogram object at subprogram elaboration is also probably more work than when using displays or static links. However, in most cases subprograms will be elaborated only once, not upon *every* subprogram call (see §4.8.5.1).

Similarly, with relay sets, when a subprogram is invoked its relay table must be copied into the activation record. With displays and static links only a single pointer must be set when a subprogram is invoked.

With relay sets two dereferences must be made to access a non-local variable. Displays only require a only single dereference to access a non-local variable (assuming display pointers are kept in registers). With static links many dereferences may be needed, for they require a search via the static links down the stack.

Storage Costs

When using static links we need to store a pointer in each activation record and to store the stack pointer in a register. When using a global display we need to store a pointer in each activation record and to store the global display in registers. When using relay sets we need to store a relay table in the subprogram template and activation record, and to store the stack pointer in a register

With relay sets both local and non-local references are efficiently resolved using a single register (that holds the stack pointer). The (run-time) calculation of the addresses of the relay table of the subprogram when it is elaborated can be thought of as performing the first dereference via the display to each non-local variable initially, and making a local copy of the actual address of the variable. However, all of the variables referenced within a subprogram must be preprocessed in this manner, not only those referenced by the subprogram

General Remarks

Indeed, the main concern with relay sets is that a very inner subprogram r will reference a large amount of data declared by an outer subprogram o. All of the intermediate subprograms from o to i will have to include the addresses of this data in their relay sets. It is possible that a compiler could recognize such subprograms (*relay subprograms*) and instead of passing relay tables could pass pointers to o's relay table. In general, relay sets are not very amenable to compiler detected optimizations since every variable in the relay table is used (when it is passed to an inner subprogram if nowhere else). Whereas, a strong argument in favor of using displays is that we tap into a large body of results on their efficient implementation.

An advantage of relay sets over displays is that subprogram parameters pose no special problems. The relay set of a subprogram completely captures its environment. While the display of a subprogram also captures its environment, when a subprogram that is a formal parameter is invoked we cannot simply replace a display pointer. The entire global display must be saved and replaced with the display of the subprogram.

If the relay sets of subprograms are small then the disadvantages noted above associated with using relay sets will not be too severe. It is in the spirit of Ada, and structured programming in general, that most objects referenced in a subprogram will either be local or passed as parameters. If subprograms adhere to these principles their use of non-local variables (and hence, the costs of using relay sets) should be low. Indeed, a study conducted by Magnusson of the reference patterns of a wide variety of SIMULA 67 program [Magnusson 1982] found that 80% of all references were to local or static variables. An additional 17% were to variables declared in the immediately surrounding scope. Thus, there is reason to believe that the size of relay sets will be small.

4.8.9.2 Costs for Tasks

We will compare the costs of using relay tables (only for tasks), shared tables (and a local stack) with the cost of using multiple cactus stacks. The results of this comparison are

	Display	Shared Table	Relay Table
Number of Stacks per task	SM	2	1
Number of Heaps (for this analysis)	0	SM - 1	SM - 1
Number of Dedicated Registers	SM	2	1
Number of levels of indirection to reference shared data	1	2	2
Extra Storage per Task	LNL + GNL*(SM-1)	GNL	LNL + SVR
Amenable to Compile Time Optimizations	yes	yes	probably not

Table 4.8 2: Comparison of Stack Management Schemes for Tasks.

SM is the number of storage modes.
GNL is the number of static nesting levels including enclosing tasks
LNL is the number of static nesting levels within a task.
SVR is the number of shared variables referenced within a task.

summarized in table 4.8.2.

Compile-Time Costs

Modified relay sets are necessary to calculate the shared set of a task in all three cases, so the compile time costs are comparable for all three schemes

Run-Time Costs

Supplying the addresses for the relay table of a task object when the body of the task is elaborated at run time is more work than what must be done for either shared tables or multiple displays. Note, however, that in most cases a task body will be elaborated only once – even when there are many task objects of the corresponding task type (See §4.8.5.3).

With relay tables, when a task is activated, its relay table must be copied into its activation record. With shared tables, two displays and activation records must be set up. With multiple cactus stacks a display and activation record must be set up for each storage mode supported.

Two dereferences must be made to access shared variables via relay or shared tables, while only one level of indirection is necessary to access shared variables via the display when using multiple cactus stacks. However, since the addresses stored in the relay and shared tables are read-only (after being written), these addresses can be cached or kept in registers obviating the second level of indirection.

Storage Costs

When using relay tables we need to store a relay table in each activation record. For the shared table, only the shared table display pointers need to be stored (even when task are not migratory). If multiple cactus stacks are used, then multiple displays will have to be stored. The storage cost of using relay tables vs shared tables is the number of shared variables vs that of static nesting levels times (both of which should be small). The storage costs of using multiple cactus stacks is the number of static nesting levels times the number of storage modes.

As was true for subprograms a clear advantage of using relay tables is that only a single register is need to efficiently access local and non-local variables. For both the relay table and display only a single dereference need be made to access shared variables, whereas for the shared a table two dereferences are required We see a clear space/time tradeoff between relay and shared tables: relay tables use extra storage to keep local copies of the addresses of shared variables, while shared tables use an extra level of indirection.

It can be argued that only a single dereference will be needed after the first reference to a shared variable with shared tables, as the address of the shared datum will vary likely be stored in a register; however, because of the low degree of temporal and spacial locality of shared data [McAuliffe 86] and the high degree of temporal and spacial locality of private data it may be more advantageous to keep a different addresses in the register (since private data is accessed more frequently).

General Remarks

Another clear advantage of relay tables for tasks is that executing an accept statement in the environment of the task which declares the entry is simplified: the environment is captured in the modified relay set of the accept. While associating a display with an accept statement would also capture its environment, the display of the accept statement may include scopes that are not in the local display of the calling task. Thus, the local display of the calling task would have to be saved and replaced with the display of the accept statement when the entry was called.

Clearly, the best two implementation for task are the relay table and shared table. Which scheme is better will depend on how many non-local variables are referenced by tasks and their nesting levels

If tasks adhere to the principles of structured programming their use of non-local variables should be low This, of course, assumes that an efficient implementation of the rendezvous can be found so that tasks will not rely heavily on shared variables for synchronization This was the intention of the language designers

5. Optimizations and Machine Dependent Support

In the previous two chapters, we described SMARTS's environment and highly parallel implementation of the tasking features of Ada. In this chapter, we consider what additional compile-time and run-time support should be provided by an RTS to allow Ada programs to make effective use of highly parallel machines.

Ada includes mechanisms, pragmas and library packages, for tailoring a compiler to its environment. By providing the appropriate interface using these mechanisms, an implementation can enable Ada programs to take advantage of novel hardware features (such as fetch&add and fetch&store). Alternatively, such features can be exploited by an implementation by means of automatic translation schemes: the compiler can recognize programming idioms that can be translated into code that makes use of the features. Other less architecture-specific compiler optimizations can also be performed to speed-up the code generated from Ada programs.

Of the three schemes (implementation-defined pragmas, library packages, and automatic optimizations), automatic optimizations is the most appealing since it requires no assistance from the programmer and does not adversely affect the portability of programs. However, automatic optimizations require a sophisticated compiler and may be untenable when program units are compiled separately. If an Ada-only implementation for a library package is supplied (e.g., the package does not use the predefined pragma INTERFACE or the predefined library package MACHINE_CODE), then programs that use the package will also be portable. Obviously, the programs will run less efficiently on implementations targeted to architectures that do support the features.

In the next section we consider some simple optimizations that could be performed by an Ada compiler. We then consider in more detail the merits of various Ada-only library package implementations for allowing Ada programs to use fetch&add's and fetch&store's

directly. Since none of these implementations is entirely satisfactory, we finish the section by briefly considering an implementation that resorts to using the package MACHINE_CODE.

5.1 Optimizations

As discussed in §4.8, performing automatic optimizations on Ada programs is exacerbated by the interaction of implicit shared variables and separate compilation. In this subsection, we sketch some simple optimizations that can be used in the presence of separate compilation and that do not entail sophisticated data or control flow analysis.

Usage Subsets

The cost of implementing many of Ada's features is driven up by their interaction with other features. This is especially true of the tasking features of Ada. For instance, to avoid asynchronously wresting control away from a task when it is aborted, many implementations of Ada (including SMARTS) require that tasks check if they have been aborted at an abort synchronization point. Thus, the inclusion of the abort statement in Ada increases the cost of implementing every Ada feature that involves an abort synchronization point (which is nevertheless regarded as less expensive than asynchronously wresting control away from a task). It has therefore been suggested that an Ada implementation provide different RTS's for subsets of Ada [Brosgol 1988]. If a compiler can ascertain that a program uses only the features in a given subset, then the RTS for that subset will be sufficient for the program. When run-time support for Ada programs is implemented by inserting calls to the RTS into Ada programs (as is done for SMARTS), then the determination of what version of the RTS is required can be made at bind time (by having the binder install the appropriate RTS).

While only one version of SMARTS currently exists, it would not be difficult to remove support for sets of features. For instance, care was taken in designing SMARTS so that every statement where the *abnormal* flag of a task is tested can be removed without affecting the integrity of the rest of SMARTS. More importantly, a streamlined version of SMARTS that did not provide support for rendezvous, delays, pre-emptive scheduling etc., could be used for programs in which all tasks run until completion, thereby reducing the weight of Ada tasks even further.

Automatic Translation Schemes

Automatic translation schemes were suggested early on as a means of reducing the cost of implementing complex Ada features. Probably the best known of these schemes is a transformation that was first proposed by Habermann and Nassi that allowed an entry caller to execute a rendezvous [Habermann and Nassi 1980]. With this transformation a rendezvous would require only a single task context switch. Unfortunately, there is a large overhead associated with this scheme due primarily to nested rendezvous, aborts and shared variables. The most severe of these impediments to an efficient implementation is the use of shared variables, that is, the requirement that accept statements be executed in the environment of the owner. As discussed in §4.8.10, when modified relay sets are used, executing an accept statement in the environment of the owner poses no special problems.

However, since SMARTS uses micro-tasking, context switches are not very costly. The overriding cost occurs when the private variables of the task that is being switched out must be flushed from the cache (note that synchronous shared variables must be flushed whether its is the caller or the owner who executes the rendezvous). Moreover, even if SMARTS were to support this translation scheme, it would not prevent rendezvous from being a bottleneck when used to coordinate large numbers of tasks. Thus, it is not clear that allowing the caller to execute rendezvous would greatly benefit an implementation of Ada on a highly parallel machine.

In the next section, we consider a rendezvous translation scheme that is more appropriate for our machine model: enabling Ada programs to use fetch&add's and fetch&store's. Library packages are used to achieve this goal, as an sophisticated compiler would be required to automatically detect when this non trivial transformation could be performed.

Since fetch&ϕ's are combined in the network of our target machine, these rendezvous will not result in serial bottlenecks even when a large number of tasks are involved.

5.2 Library Packages

To write an Ada-only package that provides procedures that have the same semantics as fetch&add or fetch&store, and hence, that can be translated directly into a single fetch&add or a fetch&store instruction, seems deceptively simple. It is, in fact, impossible to provide an Ada-only implementation for an unbounded number of tasks. An obvious implementation would seem to be to use a rendezvous with an implementation-defined task to ensure mutual exclusion while the add or swap operation is being performed. However, as we shall show in this section, tasks and rendezvous carry with them undesirable semantics. Ada shared variables can be used to implement the required critical sections, but only for a bounded number of tasks. We will consider three approaches. The first transforms a task into a record on which fetch&add's can safely be performed; thus a task is created for each variable. The second uses rendezvous with a single task to enforce the atomicity of a fetch&add operation. The third attempts to enforce atomicity using shared variables. (The case for fetch&store is analogous, and is therefore omitted.)

Translating an Active Object into a Passive Object

In [Schonberg and Schonberg 1985], a *beacon* task is defined whose rendezvous can be transformed into fetch&add's on a component of a record (The necessity of using an active object, i e , a task, to implement a passive object, such as, an integer, is unfortunately, a common occurrence in Ada due to the inability to name composite objects in a pragma SHARED or to have a task wait on multiple entry calls) A beacon task has entries for performing reads, writes, and fetch&add's on a variable that is local to the task The definition of a *beacon* task is given below

```
-- Task type for beacon task.
task type beacon is
    -- Entry for reading a shared variable.
    entry read(I : out integer);
    -- Entry for writing a variable in global memory.
    entry write(E : in integer);
    -- Entry for performing, I := fetch&add(V, E).
    entry fetch_and_add(I : out integer; E : in integer);
end beacon;

-- Task body for beacon.
task body beacon is
V : integer;
begin
    loop
        select
            -- V := E;
            accept write(E : in integer) do
                V := E;
            end write;
            or -- I := V;
                accept read(I : out integer) do
                    I := V;
                end read;
            or -- I := fetch&add(V, E);
                accept fetch_and_add(I : out integer; E : in integer) do
                    I := V;
                    V := V + E;
                end fetch_and_add;
            or -- I := fetch&store(V, E);
                accept fetch_and_store(I : out integer; E : in integer) do
                    I := V;
                    V := E;
                end fetch_and_store;
            or -- Terminate when there are no more customers.
                terminate;
            end select;
    end loop;
end beacon;
```

A beacon task must be transformed into a record containing several fields rather than a single integer to ensure that the semantics for Ada tasks are respected. For instance, if a task was actually being used, then other tasks could abort the task; hence, the record must have fields where the status of the task can be recorded Below we give the record type generated for a *beacon* task type.

```
-- Record created from beacon task type.
type BEACON_STRUCT is
    VALUE : integer,
        INITIALIZED : boolean := false;
        COMPLETED : boolean;
        INITSYNCH : integer := 0;
    end record;
```

Since *v* is local to the task body of *beacon*, it can only be accessed using the entries of the *beacon* task

The code that must be executed for a call to the entry *fetch__and__add* of a task *t* of type *beacon* is:

```
-- Code executed for a call to T.fecth_and_add(V, E) where T is of a beacon task.
    while not T.INITIALIZED or T.COMPLETE loop
        if T.COMPLETE then
            raise TASKING_ERROR;
        end if;
    end loop;
    fetch_and_add(V, E); -- Could be executed as a single fecth&add instruction.
```

As can be seen, the rendezvous cannot be executed as a single fetch&add (we must first ensure that *t* is callable).

Encapsulate Control Task in Package Body

To avoid having to do extra bookkeeping, we can encapsulated a task in the body of package whose specification declares procedures for the fetch&add's – since other task cannot see the package task, they are unable to abort it. The procedures will use rendezvous with the package task to ensure that the fetch&add operations are performed as atomic actions. The package *memory__manager* given below uses this technique and is similar to a package developed by Mitsolides for expressing fetch&add's in Ada [Mitsolides 1988].

```
    package GLOBAL_MEMORY is
        type SHARED_INTEGER is access integer,
           Procedure for reading a shared variable.
        procedure read(V : in SHARED_INTEGER) return integer.
           Procedure for writing a variable in global memory
        procedure write(V : in out SHARED_INTEGER; E in integer).
           Procedure for performing, I := fetch&add(V, E)
        procedure fetch and add(V : in out SHARED_INTEGER. E   in integer)
                    return integer
    end package
```

```
package body GLOBAL_MEMORY is
    type SHARED_INTEGER is access integer;
    -- Global memory integer synchronizer task specification, i.e., the interface
    -- it presents to other program units.
    task type integer_synchronizer is
        -- Entry for reading a shared variable.
        entry read(I : out integer; V : in SHARED_INTEGER);
        -- Entry for writing a variable in global memory.
        entry write(V : in out SHARED_INTEGER; E : in integer);
        -- Entry for performing, I := fetch&add(V, E).
        entry fetch_and_add(I : out integer; V : in out SHARED_INTEGER; E : in
        integer);
    end;

    -- A task for synchronizing integer references to global memory.
    integer_memory_monitor : integer_synchronizer;

    procedure read(V : in SHARED_INTEGER) return integer is
    -- Procedure for reading a shared variable.
        I : integer;
    begin
        integer_memory_monitor.read(I, V);
        return I;
    end read;

    procedure write(V : in out SHARED_INTEGER; E : in integer);
    -- Procedure for writing a variable in global memory.
    begin
        integer_memory_monitor.write(I, V);
    end write;

    procedure fetch_and_add( V : in out SHARED_INTEGER; E : in integer)
    return integer is
    -- Procedure for performing, I := fetch&add(V, E).
        I : integer;
    begin
        integer_memory_monitor.fetch_and_add(I, V, E);
        return I;
    end read;

    -- Global memory integer synchronizer task body, i.e., its implementation.
    task body integer_synchronizer is
    begin
        loop
            select
                -- V.all := E;
                accept write(V : in out SHARED_INTEGER; E : in integer) do
                    V.all := E.
                end write;
            or -- I := V.all;
                accept read(I : out integer; V : in SHARED_INTEGER) do
                    I := V.all;
                end read;
            or -- I := fetch&add(V, E);
                accept fetch_and_add(I : out integer; V : in out SHARED_INTEGER;
                                       E : in integer) do
                    I := V.all;
                    V.all := V.all + E;
                end fetch_and_add;
            or -- Terminate when there are no more customers.
                terminate;
            end select;
        end loop;
    end integer_synchronizer;
    begin -- Body of global_memory.
        null;
    end;
```

This solutions looks attractive on the surface: the package procedures can operate on both synchronous and asynchronous shared variables, as the rendezvous with the package task ensures the CREW access of the designated objects of the actual parameters between synchronization points; further, the fetch&add operations are performed during a rendezvous so their atomicity is guaranteed, and finally, the task is not visible outside of the package, and hence, cannot be aborted.

Regrettably, as pointed out by Dewar [Dewar 1988b], these procedures cannot be translated into a single fetch&add. Recall that the synchronous shared variables of a task must be flushed from the cache at shared variable synchronization points and that the start and end of a rendezvous are shared variable synchronization points. Since the procedures are implemented with rendezvous, the synchronous shared variables of any task that calls the procedures must be flushed. Thus, the procedure *fetch__and__add* must be translated into the sequence

```
    flush(marked);
    fetch&add(V, E);
```

(Note that the synchronous shared variables must also be flushed from the cache when beacon tasks are used.)

However, if accesses to shared variables are restricted to the package procedures, then all synchronous shared variables will be read only (when they are passed to the procedures), and hence, need not be flushed. In this case, the procedures can be transformed into a single instruction. While SMARTS adheres to this principle of only accessing synchronous shared variables via the package procedures, not all Ada programs will conform.

Shared Variables

An alternative implementation suggested by Dewar is to guarantee the atomicity of the procedures by protecting them with a synchronization mechanism based on shared variables [Dewar 1988b]. Since rendezvous would not be employed, synchronous variables would not have to be flushed from the cache, and the procedure could translated into a single fetch&add. On the other hand, the lack of synchronization points restricts the variables upon which the procedures can operate to those named in pragma SHARED. Furthermore, it is not known how to solve the general mutual exclusion problem using shared variables without some form of read-modify-write instruction. Ada does not include such an instruction – only reads and writes to variables named in a pragma SHARED are required to be atomic. However, Dekker's algorithm uses only atomic reads and writes to provide mutually exclusive access to a critical section for a known number of tasks. Thus, if we are willing to limit the number of tasks that perform fetch&add's, then Peterson's algorithm [Peterson 1981] can be used to ensure the atomicity of the package procedures. Since the number of tasks supported by an implementation is bounded (due to storage limitation if nothing else), it is entirely reasonable to similarly limit the number of task that can perform fetch&add's.

5.3 Predefined Package MACHINE_CODE

Implementing the package procedures using the predefined package MACHINE_CODE, while non-portable, is straightforward since the machine instructions themselves can be executed Ada implementations on machines without hardware support for fetch&add's could supply similar packages using the package MACHINE_CODE: whatever read-modify-write instruction was supported by the machine could be used to ensure the atomicity of the procedures

```
-- Example of procedure that uses the package MACHINE_CODE to implement fetch&add's
-- directly.
procedure machine_code_faa(V : in SHARED_INTEGER; E : in integer; I : out integer)
is
    use MACHINE_CODE;
begin
    SI_FORMAT(CODE=>fetch&add, ADDRESS => V, OPERAND => E, RESULT => I);
end fetch_and_add;
```

6. Conclusion

Experience with large-scale multiprocessors has shown that no one programming model is optimal for all applications [LeBlanc *et al.* 1988]. Accordingly, SMARTS has been designed to support a wide variety of programming styles, e.g., tasks that rely on shared variables for communication and synchronization as well as tasks that communicate and synchronize via rendezvous. That SMARTS supported a wide variety of programming styles would be moot, however, if SMARTS did not do so effectively. In this chapter, we consider how successful SMARTS is at allowing Ada programs to realize the potential of highly parallel machines. Specifically, we assess our implementation in terms of our goals of efficiency, high degree of parallelism and minimal critical sections. We conclude the chapter with a brief discussion of some possible scheduling enhancements that would allow SMARTS to make more effective use of the hierarchical memory of our target machine.

6.1 Efficiency

Although SMARTS is targetted to a multiprocessor, it is as efficient as its immediate predecessor that is targetted to a uniprocessor: it contains roughly the same number of lines of code and executes at roughly the same speed as the Ada/Ed tasking module designed by Rosen [Rosen 1985]. (Indeed, after being stripped of comments, the lengths of the two tasking modules are within 2 lines of each other: 1,800 vs 1,802 lines of C code). Thus, we lost no efficiency by moving away from a centralized supervisor based on locking to a distributed supervisor based on the self-service paradigm.

Furthermore, since the SMARTS processors self-schedule the Ada tasks created from a program, the tasks will not incur the overhead of full-blown operating system processes. Hence, we will not have to conclude "that tasks are not lightweight constructs that can be casually invoked," as Newton does about his implementation of Ada tasks using MACH threads [Newton 1987].

Finally, Ada shared variables are identified and efficiently stored in a hierarchy of memories using modified relay and shared sets. This efficient shared-variable handling, taken together with the translation schemes described in Chapter 5, enable Ada programs to coordinate large numbers of tasks in a bottleneck-free manner.

6.2 High Degree of Parallelism

The highly parallel queues employed by SMARTS similarly allow the SMARTS processors to provide RTS functions to Ada tasks in a bottleneck-free manner. By placing a single *SMARTS_node* on a ready queue, the creation, activation and termination of a large array of tasks is performed in parallel by the SMARTS processors.

The implementation of time management is serialization-free in the sense that a task can be removed from a time chain while other tasks are being enqueued to or dequeued from the time chain by a SMARTS processor. Further, each SMARTS processor can perform enqueues to and dequeues from their time chains in parallel. The only serialization in the implementation of rendezvous management is while a task attempts to rendezvous with the callers of one of its entries (note that the callers of other entries can proceed in parallel).

The implementation of aborts contains the sole serialization point of SMARTS, however, aborts are meant to be used only in abnormal situations. Hence, this serialization point should not create a bottleneck in normal Ada programs.

Our highly parallel implementation of the Ada tasking model makes it possible for computationally intensive numeric and symbolic applications to be written using Ada's rich set of tasking features without sacrificing performance. Thus, these applications can benefit from Ada's support for programming in the large while harnessing the power of large scale parallel machines.

6.3 Minimizing Critical Sections

To allow real-time applications written in Ada to make effective use of large-scale parallel machines, SMARTS minimizes response time, that is, the time for a newly executable task to pre-empt the SMARTS processor of a task of lower priority.

The responsiveness of SMARTS to a task of priority p will be bounded by the sum of four terms: the time to execute the longest processor critical section, the time slice, the time to execute the timer interrupt handler, and the time to (attempt to) dequeue a task from $max\_priority - p + 1$ ready queues. The time slice, timer interrupt handler, and ready dequeue operations are all unavoidable overheads to pre-empting a SMARTS processor.

The only processor critical sections in SMARTS are queue operations that either disable interrupts or entail busy-waiting. In SMARTS, these are limited to time chain operations, ready queue operations, and entry queue operations. The time to perform a time chain dequeue and a ready dequeue is already included in the above sum, so the time chain enqueue operation, the ready enqueue operation, and the entry queue operations are the only nonessential contributors to the response time of SMARTS.

It is shown in Appendix A that (with an admittedly ideal implementation) the response time for a task of maximal priority can be as low as 100 microseconds on a machine with an instruction cycle of 100 nanoseconds using the default time slice of 50 microseconds.

6.4 Enhancements

In SMARTS, Ada task are migratory: a task is as likely to run on one SMARTS processor as any other. To make more effective use of the local memories of processors, tasks could be given *affinity* for the SMARTS processor that is running on the processor whose local memory contains the private data of the task. Tasks that share data could have affinity for the SMARTS processor that is running on the processor whose shared portion of local memory contains the shared data.

To implement such affinity scheduling, we could partition the tasks created from an Ada program into chunks based on their mutual use of data, and then assign a chunk to the SMARTS processors where its data are stored. However, affinity scheduling introduces the possibility that the SMARTS processors will have unbalanced load of tasks. Thus, to minimize the overall execution time, we must chunk tasks together that share data, while assigning chunks to SMARTS processors so as to maintain a balanced load.

Proposed partitioning and scheduling paradigms range from static (compile-time) to dynamic (run-time). Static partitioning and scheduling may suffer because decisions must be based on imperfect knowledge and because they cannot adapt to run-time anomalies; dynamic partitioning and scheduling may suffer because of the overhead associated with chunk creation and allocation. (The current scheduling mechanism of SMARTS is an example of dynamic partitioning and scheduling with a chunk size equal to one.)

The amount of analysis used to determine the partitioning and scheduling of tasks also varies from exhaustive to none. Unfortunately, exhaustive analysis, i.e., statically determining the partitioning and scheduling of tasks that minimizes the overall time, is intractable, and thus is not practical when there are large numbers of task and processors. Hence, the affinity based scheduling of Ada tasks on SMARTS processors may have to rely on heuristics.

Appendix A – Responsiveness of SMARTS

We define the responsiveness of a RTS to be the time for a newly executable task of priority p to begin execution in the absence of other newly executable tasks. In SMARTS, the responsiveness of a task of priority p will be bounded by the sum of four terms:

(1) the time to execute the longest processor critical section;

(2) the time slice;

(3) the time to execute the timer interrupt handler; and

(4) the time to (attempt to) dequeue a task from $max_priority - p + 1$ready queues.

The processor critical sections of SMARTS are the time chain operations, the ready queue operations, and the entry queue operations. The ready and time chain dequeue operations are already included in the above sum.

The code for each of (*1*), (*2*) and (*4*) is less than 30 Ada statements that consist primarily of simple conditionals, reads, writes and fetch&ϕ's. The time chain procedures and the ready dequeue procedure contain while loops. However, we do not need to take into account the time spent spinning on *firstmult* of the ready dequeue operation because a SMARTS processor only spins while there are tasks being dequeued by other SMARTS processors. We can also discount the time spent dequeuing tasks from time chains, as this creates newly executable tasks (which are excluded in our working definition of response time). The time chain enqueue procedure is more problematic. Its execution time will depend on the duration of the delay and the number of tasks on the time chain, neither of which can be quantified a priori.

We will assume an ideal implementation where all procedure calls are inlined and where the timer functions provided by the kernel do not entail the overhead of operating systems calls. This is the case when SMARTS is running on a bare machine. Given such an implementation, we estimate that less than 100 machine code instructions would be generated for each of *(1)*, *(2)* and *(4)*. For concreteness we assume that an instruction on our target machine takes 100 nanoseconds, and we use the default time slice of 50 microseconds. Our sum can then be formulated as:

$$10 + 50 + 10 + 10*(\text{MAX\_PRIORITY} - \text{P} + 1) \text{ microseconds.}$$

So the response time for a task of maximal priority is less than 100 microseconds.

REFERENCES

Acetta *et al.* 1986

Acetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: a new kernel foundation for UNIX development," *Proceedings of the USENIX 1986 Summer Technical Conference*, pp. 93-112, 9-13 June, 1986.

Ada Issues

The approved Ada language commentaries, Grebyn Corporation, Vienna, Virginia, 1988.

Aho 1986

Aho, A. V., R. Sevi, and J. D. Ullman, *Compilers principles and tools,* Adison-Wesley, 86.

Allen *et al.* 1987

Allen, F. E., M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An overview of the PTRAN analysis system multiprocessing," *Proceedings of the 1987 International Conference on Supercomputing. Athens, Greece*, 1987.

Andrews and Schneider 1983

Andrews, G. R. and F. B. Schneider, "Concepts and notations for concurrent programming," *Computing Surveys*, vol. 15, pp. 3-43, March, 1983.

Apt 1986

Apt, K. R., "Correctness proofs of distributed termination algorithms," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 388-405, 1986.

Ardö 1984

Ardö, A., "Experimental implementation of an Ada tasking run-time system on the multiprocessor computer CM*," *Proceedings of the 1st Annual Washington Ada Symposium*, pp. 145-153, 25-27 March, 1984.

Ardö 1988a

Ardö, A., "Hardware support for execution of Ada tasking," *Proceedings of the Twenty-First Annual Hawaii International Conference on Software System Sciences*, pp. 194-202, 5-8 January, 1988.

Ardö 1988b

Ardö, A., Personal Communication, May, 1988. On hardware support for Ada rendezvous.

ARTEWG 1986

Ada Runtime Environment Working Group, "A catalogue of Interface features for the Ada run time environment," *ACM SIGAda*, 1986.

Baker 1987

Baker, T. P., "A low-level tasking package for Ada," *Using Ada: ACM SIGAda International Conference*, pp. 141-146, 8-11 December, 1987.

Bal *et al.* 1988

Bal, H. E., J. G. Steiner, and A. S. Tanenbaum, "Programming languages for distributed systems," Vrije Universiteit Rapport IR-147, February, 1988.

Baron *et al.* 1986

Baron, R. V., R. F. Rashid, E. Siegel, A. Tevanian, and M. W. Young, "MACH-1: a multiprocessor oriented operating system and environment," *SIAM*, 1986.

BBN 1986

BBN Laboratories, "Butterfly parallel processor overview," Report 6148, Version 1 , BBN Laboratories, Cambridge, Mass., March 6, 1986.

Belloch 1986

Belloch, G., "Parallel prefix versus concurrent memory access," Thinking Machines, Cambridge, Mass. Technical Report, 1986.

Bennett 1987

Bennett, J. K., "The design and implementation of Smalltalk," *OOPSLA '87 Proceedings*, pp. 318-330, 4-8 October, 1987.

Bisiani and Forin 1987a

Bisiani, R. and A. Forin, "Architectural support for multilanguage parallel programming on heterogeneous systems," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 21-30, 5-8 October, 1987.

Bisiani and Forin 1987b

Bisiani, R. and A. Forin, "Multilanguage parallel programming," *Proceedings of the International Conference on Parallel Processing*, pp. 381-384, 17-21 August, 1987.

Brantley *et al.* 1985

Brantley, W. C., K. P. McAuliffe, and J. Weiss, "RP3 processor memory element," *Proceedings of the International Conference on Parallel Processing*, 20-23 August, 1985.

Brinch Hansen 1973

Hansen, P. Brinch, *Operating system principles*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

Brosgol 1988

Brosgol, B. J., "International workshop on real-time Ada issues: summary report," *Ada LETTERS*, vol. VIII, pp. 91-107, January/February, 1988.

Burger and Nielsen 1987

Burger, T. M. and K. W. Nielsen, "An assessment of the overhead associated with tasking facilities and task paradigms in Ada," *Ada LETTERS*, vol. VII, pp. 49-58, January/February, 1987.

Burns 1987

Burns, A., "Using large families for handling priority requests," *Ada LETTERS*, vol. VII, pp. 97-104, January, February, 1987.

Buroff 1977

Buroff, S. J., "Algol 68 implementation techniques," Ph.D. Thesis, Illinois Institute of Technology, 1977.

Carnevali *et al* 1986

Carnevali, P., M. Vitaletti, and V. Zecca, "Microtasking on IBM multiprocessors," IBM European Center for Science and Engineering Computing, Rome Italy, Report Number G513-4091, April, 1986.

Carriero 1987

Carriero, N., "The implementation of tuple space machines," Ph.D. Thesis, Yale University, December, 1987.

Chen and Ci 1987

Chen, H. and Y. Ci, "Parallel execution of non-do loops," *Proceedings of the International Conference on Parallel Processing*, pp. 512-516, 17-21 August, 1987.

Cheriton 1984

Cheriton, D. R., "The V kernel: a software base for distributed systems," *IEEE Software*, vol. 1, pp. 19-43, February, 1984.

Cheriton 1985

Cheriton, D. R., "Preliminary thoughts on problem oriented shared memory: a decentralized approach to distributed systems," *Operating Systems Review*, vol. 19, pp. 26-33, October, 1985.

Clemmensen 1982
Clemmensen, G. B., ''A formal model of distributed Ada tasking,'' *Proceedings of the AdaTEC Conference on Ada*, pp. 224-237, October, 1982.

Cray 1986
Cray Research Inc., ''Cray computer systems multitasking users guide,'' Publication Number SN-0222, March, 1986.

Cytron 1985
Cytron, R., ''Useful parallelism in a multiprocessing environment,'' *Proceedings of the International Conference on Parallel Processing*, 20-23 August, 1985.

DEC 1982
Digital Equipment Corporation, *VAX-11 architecture reference manual*, May, 1982.

Dewar 1988a
Dewar, R. B. K., Personal Communication, 1988. On rendezvous with tasks whose priority is undefined.

Dewar 1988b
Dewar, R. B. K., Personal Communication, 1988. On ensuring the atomicity of fetch&add's using shared variables.

Dimitrovsky 1988
Dimitrovsky, I., ''A portable parallel Lisp environment,'' Ph.D. Thesis, Courant Institute, NYU, 1988.

Dinning 1987
Dinning, A., ''Incorporating SMARTS into AdaEd-C,'' Ada/Ed Documentation, September, 1987.

DoD 1979
United States Depertment of Defense, ''Preliminary Ada Reference Manual,'' *SIGPLAN Notices*, vol. 16, June, 1979.

DoD 1983
United States Department of Defense, *Reference Manual for the Ada programming Language, ANSI/MIL-STD-1915*, United States Government, January, 1983.

Edler 1985
Edler, J., Personal Communication, 1985. On implementing linked-lists with fetch&store's.

Edler *et al.* 1988a
Edler, J., J. Lipkis, and E. Schonberg, ''Process management for highly parallel Unix systems,'' *USENIX Workshop on Unix and Supercomputing*, 26-27 September, 1988.

Edler *et al.* 1988b
Edler, J., J. Lipkis, and E. Schonberg, ''Memory management in Symunix II: a design for large-scale shared memory multiprocessors,'' *USENIX Workshop on Unix and Supercomputing*, 26-27 September, 1988.

Falis 1982
Falis, E., ''Design and implementation in Ada of a runtime task supervisor,'' *Proceedings of the AdaTEC Conference ACM SIGPLAN*, pp. 1-9, October, 1982.

Faulk and Parnas 1988
Faulk, S. R. and D. L. Parnas, ''On synchronization in hard-real time systems,'' *Communications of the ACM*, vol. 31, pp. 274-287, March, 1988.

Filman and Friedman 1984
Filman, R. E. and D. P. Friedman, *Coordinated computing tools and techniques for distributed software*, McGraw-Hill, 1984.

Fisher and Weatherly 1986

Fisher, D. A. and R. M. Weatherly, "Issues in the design of a distributed operating system for Ada," *IEEE Computer*, pp. 38-47, May, 1866.

Flynn *et al.* 1987

Flynn, S., E. Schonberg, and E. Schonberg, "The efficient termination of Ada tasks in a multiprocessor environment," *Ada LETTERS*, vol. VII, pp. 55-76, November, December 1987.

Freudenthal and Pezé 1988

Freudenthal, E. and O. Pezé, "Efficient synchronization algorithms using fetch&add on multiple bitfield integers," To appear, 1988.

Gelernter 1985

Gelernter, D., "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, pp. 80-112, January, 1985.

Goodenough and Sha 1988

Goodenough, J. B. and L. Sha, "The priority ceiling protocol: a method for minimizing blocking of high priority tasks," *2nd International Workshop on Real-Time Ada Issues*, May, 1988.

Gottlieb 1984

Gottlieb, A., "Avoiding serial bottlenecks in ultraparallel MIMD computers," *Proceedings of the COMPCON '84 Twenty-Eight IEEE Computer Society International Conference*, pp. 354-358, 27 February - 1 March, 1984.

Gottlieb 1987a

Gottlieb, A., "An overview of the NYU ultracomputer project," in *Experimental Computing Architectures*, ed. J. J. Dongarra, pp. 25-95, North Holland, 1987.

Gottlieb 1987b

Gottlieb, A., Personal Communication, 1987. On implementing fetch&add's with read, steal and write.

Gottlieb *et al.* 1983a

Gottlieb, A., B. D. Lubachevsky, and L. Rudolph, "Basic techniques for the coordination of of very large numbers of cooperating sequential processors," *ACM TOPLAS*, vol. 5, pp. 164-189, April, 1983.

Gottlieb *et al.* 1983b

Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU ultracomputer — designing an MIMD shared memory parallel machine," *IEEE Transactions on Computers*, vol. 32, February, 1983.

Gries 1971

Gries, D., *Compiler construction for digital computers*, Wiley, 1971.

Habermann and Nassi 1980

Habermann, A. N. and I. R. Nassi, "Efficient implementation of Ada tasks," Carnegie-Mellon University Technical Report CMU-CS-80-103, January, 1980.

Harris and Wetts 1985

Harris, L. A. and Y. Wetts, "Resetting Displays," *SIGPLAN Notices*, vol. 20, pp. 73-77, August, 1985.

Harrison 1986

Harrison, M., "The add-and-lambda operation: an extension of f&a," Ultracomputer Note #104 Courant Institute, NYU, July, 1988.

Harrison 1988

Harrison, M., "More uses for Add-and-Lambda," Ultracomputer Note #132, Courant Institute, NYU, 1988.

Haynes *et al.* 1982

Haynes, L. S., R. L. Lau, D. P. Siewiorek, and D. W. Mizell, ''A survey of highly parallel computing,'' *Computer*, pp. 9-46, January, 1982.

Hibbard *et al.* 1981

Hibbard, P. A., A. Higen, J. Rosenberg, M. Shaw, and M. Sherman, *Studies in Ada style*, 1981.

Hilfinger 1982

Hilfinger, P. N., ''Implementation strategies for Ada tasking idioms,'' *Proceedings of the AdaTec Conference on Ada*, pp. 26-30, October, 1982.

Hillis 1985

Hillis, W. D., *The connection machine*, MIT Press, Cambridge, Mass, 1985.

Hillis and Steele 1986

Hillis, W. D. and G. L. Steele, ''Data parallel algorithms,'' *Communications of the ACM*, vol. 29, December, 1986.

Hoare 1981

Hoare, C. A. R., ''The emperor's old clothes,'' *Communications of the ACM*, vol. 24, pp. 75-82, February, 1981.

Hummel and Zhang 1987

Hummel, R. A. and K. Zhang, ''Dynamic processor allocation for parallel algorithms in image processing,'' *Proceedings of the Optical and Digital Pattern Recognition session of the SPIE Conference on EO-Imaging*, vol. 754, pp. 268-275, January, 1987.

Jha and Kafura 1985

Jha, R. and D. Kafura, ''Implementation of Ada synchronization in embedded, distributed systems,'' Virginia Polytechnic Institute Technical Report TR-85-23, 1985.

Kafura 1985

Kafura, D., ''Termination of Ada tasks in a distributed environment,'' Virginia Polytechnic Institute Technical Report TR-85-22, 1985.

Kaiser 1986

Kaiser, G. E., ''Generation of Run-Time Environments,'' *Proceedings of SGPLAN '86 Symposium on Compiler Construction Conference*, pp. 51-56, 25-27 June, 1986.

Klumpp 1987

Klumpp, A. R., ''Linear algebra and other Ada packages adapted from HAL/S, Fortran, PL/I, and MAC,'' *Using Ada: ACM SIGAda International Conference*, 8-11, 1987.

Kruchten 1985

Kruchten, P., ''The Ada machine,'' Ada/Ed Documentation, September, 1985.

Kruchten and Schonberg 1984

Kruchten, P. and E. Schonberg, ''The Ada/Ed system: a large-scale experiment in software prototyping using SETL,'' *Technology and Science of Information*, vol. 3, pp. 175-181, 1984.

Lampson 1982

Lampson, B. W., ''Fast procedure calls,'' *SIGPLAN Notices*, vol. 17, April, 1982.

LeBlanc 1983

LeBlanc, T. J., ''Remote memory references in a distributed programming language,'' University of Rochester, Technical Report, August, 1983.

LeBlanc and Jain 1987

LeBlanc, T. J. and S. Jain, ''Crowd control: coordinating processes in parallel,'' *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 81-84, 17-21 August, 1987.

LeBlanc *et al.* 1988

LeBlanc, T. J., M. L. Scott, and C. M. Brown, "Large-scale parallel programming: experience with the BBN butterfly parallel processor," *Proceedings of the ACM SIGPLAN Conference on Parallel Programming Experience with Applications, Languages and Systems*, September, 1988.

Ledgard and Singer 1982

Ledgard, H. F. and A. Singer, "Scaling down Ada (or towards a standard Ada subset)," *Communications of the ACM*, vol. 25, February, 1982.

Li 1986

Li, K., "Shared virtual memory on loosely coupled multiprocessors," Ph.D. Thesis, Yale University, 1986.

Lindquist and Joyce 1985

Lindquist, T. E. and R. C. Joyce, "Ada task synchronization in a multiprocessor system with shared memory," *Journal of Pascal, Ada, & Modula-2*, pp. 9-19, January/February, 1985.

Lipkis 1985

Lipkis, J., Personal Communication, 1985. On avoiding the overhead of dynamic storage allocation.

Lipkis *et al.* 1987

Lipkis, J., E. Schonberg, and J. Edler, "Issues in the parallel runtime environment," Private Comunication, December 15, 1987.

Locke and Goodenough 1988

Locke, C. D. and J. B. Goodenough, "A practical application of the ceiling protocol in a real-time system," *2nd International Workshop on Real-Time Ada Issues*, May, 1988.

Magnusson 1982

Magnusson, K., "Identifier references in Simula 67 programs," *Simula Newsletter*, vol. 10 (2), 1982.

Maloney and Reed 1987

Malloney, A. D. and D. A. Reed, "MFP: a portable message passing facility for shared memory multiprocessors," *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 739-741, 17-21 August, 1987.

McAuliffe 1986

McAuliffe, K., "Analysis of cache memories in highly parallel systems," Ph.D. Thesis, Courant Institute, NYU, May, 1986.

Melde and Gage 1987

Melde, J. E. and P. G. Gage, "Large system simulation using Ada," *Using Ada: ACM SIGAda International Conference*, pp. 126-132, 8-11 December, 1987.

Middlemas 1987

Middlemas, M. R., "Ada applications on embedded targets," *Using Ada: ACM SIGAda International Conference*, pp. 170-179, 8-11 December, 1987.

MIPS 1987

MIPS Computer Systems Inc., *MIPS R2000 processor user's guide*, 1987.

Mitsolides 1988

Mitsolides, T., "Ada tasks in a parallel environment," New York University, Report, January, 1988.

Newton 1987

Newton, T. D., "An implementation of Ada tasking," Carnegie-Mellon University Technical Report CMU-CS-87-169, October, 1987.

Operowsky 1988

Operowsky, H. L., "Optimization and garbage collection in Ada programs on shared memory computers," Ph.D. Thesis, Courant Institute, NYU, 1988.

Peterson 1981
Peterson, G. L., ''Myths about the mutual exclusion problem,'' *Information Processing Letters*, vol. 12, pp. 115-116, June, 1981.

Pfister *et al.* 1985
Pfister, G. F., W. C. Brantley, D. A. George, S. L. Harvey, K. P. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, ''The IBM research parallel processor prototype (RP3): introduction and architecture,'' *Proceedings of the 1985 International Conference on Parallel Processing*, 20-23 August, 1985.

Ranade 1988
Ranade, A., ''Fluent parallel computation,'' Presentation at NYU, February 8, 1988.

Rettberg 1987
Rettberg, R. D., ''Monarch,'' Presentation at NYU, November 9, 1987.

Riccardi and Baker 1984
Riccardi, G. A. and T. P. Baker, ''A runtime supervisor to support Ada task activation, execution and termination,'' *IEEE Computer Society 1984 Conference on Ada applications and environments*, pp. 14-22, 14-22 October, 1984.

Roark and McAfee 1988
Roark, C. and R. McAfee, ''The applicability of Ada to MIL-STD-1750A,'' *Ada LETTERS*, vol. VIII, pp. 84-86, May/June, 1988.

Rosen 1983
Rosen, J. P., *Proceedings of the Joint Ada-Europe/AdaTEC Conference*, March, 1983.

Rosen 1985
Rosen, J. P., ''The task management system,'' Ada/Ed Documentation, September, 1985.

Rosenblum 1987
Rosenblum, D. S., ''An efficient communication kernel for distributed Ada runtime tasking supervisors,'' *Ada LETTERS*, vol. VII, pp. 102-117, March/April, 1987.

Rudolph 1982
Rudoloh, L., ''Software structures for utraparallel computing,'' Ph.D. Thesis, Courant Institute, NYU, 1982.

Russell and Waterman 1987
Russell, C. H. and P. J. Waterman, ''Variations on Unix for parallel-processing computers,'' *Communications of the ACM*, vol. 30, pp. 1048-1055, December, 1987.

Schonberg and Schonberg 1985
Schonberg, E. and E. Schonberg, ''Highly parallel Ada — Ada on an ultracomputer,'' in *Ada in use — Proceedings of the Ada-Europe International Conference*, ed. J. G. P. Barnes and G. A. Fisher, pp. 58-71, Cambridge University Press, 14-16 May, 1985.

Schultz and Chandna 1987
Schultz, W. L. and A. Chandna, ''An Ada based approach to factory scale MAP network simulation,'' *Using Ada: ACM SIGAda International Conference*, 8-11 December, 1987.

Schwartz 1980
Schwartz, J. T., ''Ultracomputers,'' *ACM Transactions on Programming Languages and Systems*, vol. 2, pp. 484-521, October, 1980.

Schwartz *et al.* 1986
Schwartz, J. T., R. B. K. Dewar, E. Dubinsky, and E. Schonberg, *Programming with sets as introduction to SETL*, Springer-Verlag, New York, 1986.

Scott 1986

Scott, M. L., "The interface between distributed operating system and high-level programming language," *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 242-249, 19-22 August, 1986.

Sebesta 1987

Sebesta, R. W., "Yet another survey of Ada usage and Ada training," *Ada LETTERS*, vol. VII, pp. 34-39, September/October, 1987.

Seidewitz 1987

Seidewitz, E., "Object-oriented programming in Smalltalk and Ada," *OOPSLA '87 Proceedings*, pp. 202-213, 4-8 October, 1987.

Seigel *et al.* 1981

H. J. Seigel *et al.*, "PASM: a parallel multimicrocomputer SIMD/MIMD system for image processing and pattern recognition," *IEEE Transactions on Computers*, vol. C-30, pp. 934-947, December, 1981.

Seitz 1985

Seitz, C. L., "The cosmic cube," *Communications of the ACM*, vol. 28, pp. 22-33, January, 1985.

Sha *et al.* 1987

Sha, L., R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," Carnegie-Mellon University Technical Report, 1987.

Shulman 1987

Shulman, N. V., "The semantics of shared variables in parallel programming languages," Ph.D. Thesis, Courant Institute, NYU, 1987.

Smith 1981

Smith, B. J., "Architecture and applications of the HEP multiprocessor computer system," *Real Time Signal Processing IV*, vol. 298, August, 1981.

Stanfill and Waltz 1986

Stanfill, C. and D. Waltz, "Toward memory-based reasoning," *Communications of the ACM*, vol. 29, pp. 1213-1228, December, 1986.

Stankovic 1988

Stankovic, J. A., "A serious problem for next-generation computers," *IEEE Computer*, pp. 10-19, October, 1988.

Stevenson 1980

Stevenson, D. R., "Algorithms for translating Ada multitasking," *Proceedings of the ACM SIGPLAN Symposium on the Ada programming language, ACM Sigplan Notices*, vol. 15, November, 1980.

Stone *et al.* 1985

Stone, J. M., F. Darema-Rogers, V. A. Norton, and G. F. Pfister, "Introduction to the VM/EPEX Fortran preprocessor," IBM Research Report RC11407, September, 1985.

Tanenbaum and Renesse 1985

Tanenbaum, A. S. and R. van Renesse, "Distributed operating systems," *Computing Surveys*, vol. 17, pp. 419-470, December, 1985.

Test 1986

Test, J., "Multi-processor management in the Concentrix operating system," *Proceedings of the USENIX 1986 Winter Technical Conference*, pp. 173-182, 15-17 January, 1986.

Volksen and Wehrum 1986

Volksen, G. and P. Wehrum, "Transition to Ada for supercomputers," in *Ada: Managing the Transition — Proceedings of the Ada-Europe International Conference*, ed. P. J. L. Wallis, pp. 79-89, Cambridge University Press, 1986.

Walters 1987
Walters, M. D., ''Expert system development in LISP and Ada,'' *Using Ada: ACM SIGAda International Conference*, 8-11 December, 1987.

Wilson 1988
Wilson, J., ''Task and memory management for MIMD shared memory parallel computers,'' Ph.D. Thesis, Courant Institute, NYU, 1988.

Wirth 1987
Wirth, N., ''Hardware architectures for programming languages and programming languages for hardware architectures,'' *Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2-7, 5-8 October, 1987.

Yemini 1982
Yemini, S., ''On the suitability of Ada multitasking for expressing parallel algorithms,'' *Proceedings of the AdaTec Conference on Ada*, pp. 91-97, October, 1982.

Yew *et al.* 1986
Yew, P.C., N. F. Tzeng, and D. H. Lawrie, ''Distributing hot-spot addressing in large-scale multiprocessors,'' *Proceedings of the international Conference on Parallel Processing*, pp. 51-58, 19-22 August, 1986.

Zeigler *et al* 1981
Zeigler, S., N. Allegre, R. Johnson, J. Morris, and G. Burns, ''Ada for the Intel 432 microcomputer,'' *Computer Magazine*, 1981.

NYU COMPSCI TR-495 c.2
Hummel, Susan Flynn
SMARTS
